

Swansea University E-Theses

An assessment of the usability of aspect-oriented programming and the usability of current implementations.

Rohani-Sarvestani, Nadim

How to cite:

Rohani-Sarvestani, Nadim (2010) *An assessment of the usability of aspect-oriented programming and the usability of current implementations..* thesis, Swansea University.
<http://cronfa.swan.ac.uk/Record/cronfa43138>

Use policy:

This item is brought to you by Swansea University. Any person downloading material is agreeing to abide by the terms of the repository licence: copies of full text items may be used or reproduced in any format or medium, without prior permission for personal research or study, educational or non-commercial purposes only. The copyright for any work remains with the original author unless otherwise specified. The full-text must not be sold in any format or medium without the formal permission of the copyright holder. Permission for multiple reproductions should be obtained from the original author.

Authors are personally responsible for adhering to copyright and publisher restrictions when uploading content to the repository.

Please link to the metadata record in the Swansea University repository, Cronfa (link given in the citation reference above.)

<http://www.swansea.ac.uk/library/researchsupport/ris-support/>

An Assessment of the usability of Aspect-oriented Programming and the usability of current implementations

by

Nadim Rohani-Sarvestani

Supervisor: Dr C.P.Jobling

A thesis submitted to the
University of Wales
in fulfilment for the degree of
MASTER OF PHILOSOPHY

School of Engineering

SWANSEA UNIVERSITY
2009



Swansea University
Prifysgol Abertawe

ProQuest Number: 10821530

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



ProQuest 10821530

Published by ProQuest LLC (2018). Copyright of the Dissertation is held by the Author.

All rights reserved.

This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346



Summary

Crosscutting concerns are responsible for producing scattered and tangled representations throughout the software life cycle. Effective separation of such concerns is essential to improve understandability and maintainability of system components at the various software development stages. Aspect-oriented software development (AOSD) holds promise for the purpose.

The study discussed how modularization can aid the development of a robust, re-usable, flexible and sustainable system. It suggests that modular programming can be achieved when certain criteria are met and, while the sustainability of modularity requires certain rules. The study introduced assumptions about software design processes and programming languages. The study recommended that a design process and a programming language work well together when the programming language provides abstraction and composition. These mechanisms can cleanly support the kinds of units the design processes that break the system into and a clear and simple one-to-one mapping from design level concepts to their source code implementation.

The study analysed the state-of-the-art in AOP techniques that would provide the tools to assess and compare AOP versus other programming approaches. It investigates language models and meta-models for AOP which would allow a more general but comprehensive comparison and analysis of the fundamental aspect language features as well as their implementation and execution techniques. It contributed to the aspect-oriented software development (AOSD) survey by classifying an aspect extension to a procedural language.

Furthermore, different scenarios were explored to understand the usability, usefulness, strengths and weaknesses of the AOP as a software technique and the current strategies that are in place to deal with crosscutting concerns. In addition, three different case studies were selected to analyse AOP implementations of none trivial applications that

uncovered benefits and drawbacks of the AOP technique. The first case study provided a comparative analysis of the changes required to evolve the tangled and scattered code versus aspect-oriented implementations. The second case study presented an AOP implementation of a crosscutting concern known as persistence and showed that persistence can be a highly re-usable aspect and be developed into a general aspect-based persistence framework. The third case study outlined how to conduct AOSD with use-cases. This contribution offered a new way of visualizing and capturing application and infrastructure use case flows while keeping infrastructure separate from the application and infrastructure services separate from each other. The use-case models that were analyzed also helped to verify that a resilient architecture is achieved by treating infrastructure use-cases as extensions of application use-cases.

Declaration/Statements

Declaration

This work has not previously been accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

Signed (N. Rohani-Sarvestani, candidate)

Date

Statement 1

This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by footnotes giving explicit references. A bibliography is appended.

Signed (N. Rohani-Sarvestani, candidate)

Date

Statement 2

I hereby give consent for my thesis, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed (N. Rohani-Sarvestani, candidate)

Date

Acknowledgements

The period of my research study and hence the completion of this thesis would not have been possible without the steady support of many special people, hopefully, most will be listed below.

First, I would like to thank my supervisor Dr C.P. Jobling for his guidance and enthusiasm on the subject.

An especial acknowledgment goes to my wife Nesrin for her love; support and patience during my candidature.

My warmest thanks go to my family Bijan, Nagme, Nada, Lua and Noura for the support throughout my education and their continual encouragement and guidance.

Finally, I would also like to thank all other supporting people that have inspired and encouraged me in some way shape or form, in particular Neil P., Hussein Y., Jonathan J., Dimitris P., Justin B., Josie A., Geoff A., Dr Tim D., Dr Andy M. and Matt J. for their friendship and support.

Contents

Summary	1
Contents	6
List of Figures	8
List of Tables.....	9
List of Abbreviations.....	10
1. Introduction	11
2. Evolution of Modularization	13
2.1 Programming Language Evolution	13
2.2 Modularization	15
2.2.1 Criteria for Modularization	17
2.2.2 Rules for Modularization	20
2.3 Constraints of Object Oriented Technologies	23
2.4 Aspect-oriented Programming	27
2.5 Summary	33
3. AOP Language Metamodel.....	35
3.1 Overview	35
3.2 Language Models	38
3.3 Survey Dimensions and Results.....	39
3.4 Common Language Concepts Metamodel	45
3.5 Execution Semantics of the Metamodel Interpreter	50
3.6 Classification of Aspect Languages According to the Metamodel.....	57
3.6 Summary	63
4. Assessing AOP – Approach and Implementation.....	66
4.1 Evaluation of Software Techniques and Management of Concerns During Evolution Tasks.....	67
4.1.1 Evaluating a Software Development Technique.....	67
Tools Used	68
Case Studies	68
Experiments.....	69
Lessons Learned.....	70
1. Selection of an Evaluation Method	70
2. Maintaining Realism	72
3. Designing the Empirical Study	72
Conclusion	72
4.1.2 Managing Crosscutting Concerns During Software Evolution Tasks	73
Setup and Tools Used	75
Results.....	76
Result Implications	79
4.2 Case Study I: A Retroactive Study of Aspect Evolution in Operating System Code	81
Overview	81
Analysis of the Crosscutting Concerns	83
1. Page Daemon Activation concern.....	83
2. Pre- fetching concern	85

3. Disk Quotas Concern	87
4. Blocking in Device Drivers Concern	88
Analysis of the Results of the Experiment.....	89
1. Evolving Scattered and Tangled Code.....	90
Page Daemon Wakeup	90
Pre-fetching	90
Disk Quotas	90
Device Blocking in Drivers.....	91
2. General Improvements using Aspects	91
3. Runtime Costs	93
Conclusion and Open Issues	94
4.3 Case Study II: Persistence as an Aspect.....	94
Modularising Persistence	96
1. Database access	96
2. SQL Translation	102
3. The Emerging Persistence Framework	105
Conclusion	107
4.4 Case Study III: AOSD with Use-Cases.....	108
4.4.1 Introduction	109
4.4.2 Solution Architecture	111
4.4.3 Capturing Concerns with Use-Cases.....	115
4.4.3.1 Requirements Gathering.....	115
4.4.3.2 Use-Case modelling	116
4.4.3.2 Use-Case Specification	119
4.4.3.4 Use-case Slices.....	122
4.4.3.5 Visualizing Use-Case Flows	123
4.4.3.6 Capturing Infrastructure Use-Cases	125
4.4.3.7 Visualizing Infrastructure Use-Case Flows	126
4.4.3.8 Analysis Model	128
4.4.3.9 Keeping Infrastructure Use-Cases Separate.....	131
4.4.3.10 Conclusion	133
4.4 Summary	135
5. Conclusion and Future Work	138
5. 1 Conclusion and Discussion	138
5. 2 Further Work.....	145
Bibliography.....	146
Appendix A	159
Appendix B	160

List of Figures

Figure 1 “one node to all” and “maximised nodes”	22
Figure 2 System evolution without modularization	26
Figure 3 System evolution: AOP Based.....	27
Figure 4 Aspects crosscut classes	31
Figure 5 AOSD Timeline [32]	36
Figure 6 Aspect Language Dimensions	40
Figure 7 Survey Dimensions and Common Metamodel adapted from [54]	46
Figure 8 The Join Point Metamodel adapted from [54]	47
Figure 9 The Pointcut metamodel adapted from [54]	48
Figure 10 The base- and aspect-level interpreters of the metamodel from [54]	56
Figure 11 AspectC in the Join point Metamodel	59
Figure 12 AspectC in the Pointcut Metamodel	61
Figure 13 Persistence framework from [80]	106
Figure 14 High level view of the current and new environment.....	110
Figure 15 User Access Control Architecture	113
Figure 16 User Access Control showing Presentation Layer Components	114
Figure 17 Self-registration process use cases.....	118
Figure 18 Pre-Registration Process.....	124
Figure 19 Registration Request.....	124
Figure 20 User Account Activation	124
Figure 21 <Perform Transaction> use-case	125
Figure 22 Structuring infrastructure use-cases.....	126
Figure 23 Handle Authorization use-case	126
Figure 24 Handle Scalability use-case	127
Figure 25 Provide Cached Access use-case	127
Figure 26 Smart Sync use-case	127
Figure 27 Analysis stereotypes	128
Figure 28 Interaction diagram for Account Activation use-case	130
Figure 29 Interaction diagram for Handle Authorization	130
Figure 30 Interaction diagram for Smart Sync.....	131
Figure 31 Infrastructure package for Handle Authorization use-case	132
Figure 32 Use-Case slice Authorization [138, p. 249].....	133
Figure 33 Execution Model Dimensions.....	160

List of Tables

Table 1 Matching join points for AspectC.....	60
Table 2 Experimental Methods Overview and Results.....	70
Table 3 Developers task descriptions, obstacles and strategies	77
Table 4 Summary of the results	91
Table 5 Functional Requirements	116
Table 6 Non-Functional Requirements	116
Table 7 Actor names and their description	118
Table 8 Self-Registration process – Use-Case 000 specification: Pre-Registration Process	119
Table 9 Self-Registration process - Use Case 001 specification: Registration Request	120
Table 10 Self-Registration process – Use-Case 002 specification: User Account Activation.....	121
Table 11 Composing peer use-case realizations with use-case slices.....	122

List of Abbreviations

AOSD: Aspect-Oriented Software Development

ECOOP: European Object-oriented Programming

ESEC: European Software Engineering Conference

FOOL: Foundations of Object-Oriented Languages

SIGSOFT: Symposium on the Foundations of Software Engineering

ICSE: International Conference on Software Engineering

JAOO: Java and Object-Oriented Software Engineering Conference

OOPSLA: Object-oriented Programming Systems, Languages and Applications

OO: Object-oriented

OOP: Object-oriented programming

AOP: Aspect-oriented programming

VM: Virtually Memory

OS: Operating System

CASB: Common Aspect Semantics Base

TAM: Tivoli Access Manager

TDS: Tivoli Directory Server

TIM: Tivoli Identity Manager

1. Introduction

The aim of this research is to introduce aspect-oriented programming (AOP) and its benefit when modularizing concerns. It introduces a model that would allow a more general but thorough comparison and analysis of the fundamental aspect language features, implementation and execution techniques. It contributes to the aspect-oriented software development (AOSD) survey by classifying an aspect extension to a procedural language. It suggests ways to assess AOP as a software technique and introduce non-trivial applications that applied AOP in order to strengthen the claim that this technique benefits the current conventional programming. It introduces a new way of visualizing and capturing application and infrastructure use case flows. Finally, it discusses any drawbacks that were found from the results of various experiments and case studies from the AOSD community.

The main part of research is divided in the following chapters:

Chapter 2 discusses how modularization can aid the development of a robust, re-usable, flexible and sustainable system. It includes a survey of programming language evolution, introduction of the concept of modularization, the principles required when decomposing a system into modules, discussion regarding the constraints of the object-oriented approach when capturing or implementing modularity concepts, and finally introducing the aspect-oriented approach and its benefits.

Chapter 3 continues the discussion regarding that AOP provides support for design decisions are difficult to express cleanly in code using existing programming techniques because they crosscut the systems' basic functionality. Subsequently, it aims to reflect and analyse the state-of-the-art in AOP techniques that would provide the tools to assess and compare AOP versus other programming approaches. It investigates language models and meta-models for AOP which would allow a more general but comprehensive comparison and analysis of the fundamental aspect language features as well as their implementation and execution techniques.

In this chapter the study also contributes to the survey by the classification of a simple AOP extension to the programming capabilities of C. Modelling AspectC would assist in a better understanding of AOP capabilities and constraints, when trying to facilitate an AOP implementation in a procedural language. Modelling will also aid the understanding of a case study that is analysed in chapter 4.

Chapter 4 assesses AOP as a software technique and introduces a benchmark that any technique must meet. The chapter begins by reporting the results of the research of two papers that discuss the evaluation of a new software development technique in terms of its usability, usefulness, strengths and weaknesses of the AOP methods and the current strategies that are in place in order to deal with crosscutting concerns.

Finally, three different case studies were selected to analyse real world non trivial applications discussing the benefits and drawbacks of the AOP technique. The first case study provides a comparative analysis of the changes required to evolve the tangled and scattered versus aspect-oriented implementations. The second case study presents an AOP implementation of a classical example of crosscutting concern known as persistence. The third case study, a new contribution towards the AOSD community, outlines how to conduct AOSD with use-case driven approach. The suggested solution is a new way of visualizing and capturing application and infrastructure use case flows while keeping infrastructure separate from the application and infrastructure services separate from each other.

Chapter 5 concludes the thesis and examines the potential of further investigations.

2. Evolution of Modularization

This chapter discusses how modularization can become the building blocks of a robust, re-usable, flexible and sustainable system. It starts with a brief survey in order to establish a perspective of the programming language evolution. Then, the concepts of modularization, its meaning, and benefits are introduced. Next, criteria and rules are discussed when decomposing a system into modules. The discussion leads to some constraints of the object-oriented approach, while capturing or implementing modularity concepts. This results into introducing the aspect-oriented approach and its benefits.

2.1 Programming Language Evolution

Assembly [1] was one of the first programming languages created for computers in the early 1950s. Soon after, FORTRAN [2], a procedural (i.e. routines, subroutines, methods), imperative programming language was developed that is especially suited to numeric computation and scientific computing. An important milestone in 1960s was the structured programming language known as ALGOL 60 (Algorithmic language) [3]. ALGOL 60 set a standard for block structure as it is known today. It supported branching, looping, delimited scope of variables, pass by value, pass by name, and recursion.

In the 1970s, Simula67 (Simulation language) [4] provided linguistic support for object-orientated programming (OOP), and CLU (function clusters) [5] provided linguistic support for data abstraction. While Simula67 supported encapsulation when developers obeyed rules, CLU offered further language enforcement, contributing to a key idea in programming methodology from the same era that focused on separation of concerns [6], organising systems into separate parts that could be dealt with in relative isolation. Although, the idea of what precisely constitutes a concern remains rather vague [7], linguistic support for modules as a collection of operations with hidden information separating the ‘what’ from the ‘how’ was standard for some time in languages such as C [8] that supported library modules with separate compilation. Also, breaking a system into modules required some criteria for decomposition. Parnas [9] originally suggested

that decomposition should begin with a list design decisions that are either difficult or likely to change, and those decisions should be hidden into modules. Parnas set some additional criteria for good modularity including support for comprehensibility and independent development which will be discussed later.

Smalltalk [10], an object-oriented reflective programming language, developed at roughly the same time as CLU, had early support for what was later called metaobject protocols [11]. Metaobjects enabled dynamic manipulation of methods or types in an application. This approach offered a powerful way of making system-wide, crosscutting changes by facilitating the modification of language implementation. Open implementation [12] allowed clients of a module to influence its implementation by use of a metaobject, accessed through a separate module interface.

The 1980s were years of relative consolidation. For example, C++ [13] combined object-oriented and systems programming or Ada [14] also an object-oriented and systems programming language intended for use by defence contractors. Therefore, instead of creating new paradigms, all of these movements elaborated upon the ideas invented previously. However, there was an increased focus on programming for large-scale systems through the use of modules, or large-scale organizational units of code. Many researchers expanded on the ideas of the existing languages and adapted them to new contexts. For example, the languages of the Argus and Emerald systems adapted OOP to distributed systems [15].

Then, in 1990s also known as the internet age, [16] more OOP languages were developed such as Java [17] that were influenced by the well established OOP principles, such as modularization mentioned by Parnas [9]. Furthermore, structuring implementations along dimensions that continue to go beyond standard procedural or object-oriented technology has been addressed by several research projects in the Aspect-Oriented Software Development research community (AOSD) [18]. Some examples are subject-oriented programming [19] and subsequent work on hyperspaces [20] which deal with collection of classes that define a view of a domain, and provide a

means of integrating these multiple views for the development of complex systems. Another notable example is aspect-oriented programming (AOP) [21], as defined by work in the AspectJ project, which provides linguistic support for concerns that are inherently crosscutting – by their very nature they are present in more than one module. The premise of this approach is that some concerns dictate a natural primary modular decomposition of a system, whereas others, called aspects, crosscut this structure. The goal is to better separate and modularise crosscutting functionality from the primary decomposition of the system using simple linguistic mechanisms. This will be analysed in more detail.

2.2 Modularization

As seen earlier the concept of modularization has been around for some time and is introduced as a mechanism for improving the flexibility and comprehensibility of a system whilst permitting curtailment of its development time [9]. Because modularization is a broad subject, the perspective of this research, when discussing modularization is assessing the benefits that AOP claim to provide and try to define the ‘ideal’ modular programming technique.

A lucid statement of the philosophy of modular programming can be found in a 1970 textbook [22] on the design of system programs by Gauthier and Pont, which states that *“a well defined segmentation of the project effort ensures system modularity. Each task forms a separate, distinct program module. At implementation time each module and its inputs and outputs are well-defined, there is no confusion in the intended interface with other system modules. At checkout time the integrity of the module is tested independently; there are few scheduling problems in synchronizing the completion of several tasks before checkout can begin. Finally, the system is maintained in modular fashion; system errors and deficiencies can be traced to specific system modules, thus limiting the scope of detailed error searching”*.

Subsequent to this statement, Parnas [9] discussed the benefits expected from modular programming (modularization) suggesting some criteria which can be used in

decomposing a system into modules. The definition of “module” is considered to be a responsibility assignment rather than a subprogram; modularizations include the design decisions which must be made before the work on independent modules can start.

Quoted benefits expected of modular programming are [9] :

- Managerial: Development time could be reduced if separate groups could work on each module with little need for communication.
- Product flexibility: The possibility of making drastic changes to one module without a need to change others.
- Comprehensibility: A system can be better designed and understood if it is possible to study it one module at a time.

Furthermore, the effectiveness of modularization is dependent upon the criteria used in dividing the system into modules. One method to decompose a system design problem is to begin with a basic flowchart and move from there to a detailed implementation. This is useful when the problem domain is for small applications. However, when the application develops and grows to a larger scale, issues such as changeability, independent development and comprehensibility become important and vital for the system to remain modular [9].

Another method is to decompose a system design problem using “information hiding” as a criterion [23]. Modules therefore no longer correspond to steps in processing but rather tend to vary as the specifications continue to change. Hence, the design begins with a list of difficult design decisions or ones which are likely to change; each module is then designed to hide such decisions from the others. Since, in most cases, design decisions transcend the time of execution, modules will not correspond to steps in processing. An example of decompositions mentioned by Parnas is the sequence in which certain items will be processed should (as far as practical) be hidden within a single module. However, various changes ranging from equipment additions to unavailability of certain resources in an operating system make sequencing extremely variable. Furthermore,

efficiency and implementation can reduce the development to a relatively independent number of small manageable programs.

Subsequently, important issues such as comprehensibility, efficiency, extensibility and reusability came into consideration and the need for flexible system architecture, made by autonomous software components became apparent [23]. Modular programming, already mentioned by Parnas, was once taken to mean the construction of programs as assemblies of small pieces; usually subroutines. But such a technique cannot bring real extensibility and reusability unless modules (i.e. a responsibility assignment rather than a subprogram) are used [9]. It is important to explore what precise properties a method must possess to deserve the modular label. Focusing on subjects such as; design methods, early stages of system construction (analysis, specification), implementation and maintenance, will provide a better understanding of object technology and refine this informal definition of modularity [24].

Next, the effectiveness of a modularization is dependent upon the criteria and rules used in dividing the system into modules. Therefore, some criteria and rules of modularity extending Parnas' principles are introduced which; taken collectively, cover the most important requirements of a modular design method.

2.2.1 Criteria for Modularization

Five fundamental design requirements need to be satisfied for a design method to be called modular [25]. These are:

a) Modular decomposability

The Modular decomposability criterion is satisfied when a software construction helps in the task of decomposing a software problem into a small number of less complex sub-problems, connected by a simple structure, and independent enough to allow further work to proceed separately on each of them [25, p. 40]. A natural effect of the decomposability requirement is division of labour: once the system is decomposed into subsystems, work allocation should also be distributed among the different systems.

This is a difficult task since it limits the dependencies that may exist between the subsystems. Therefore, such dependencies must be kept to bare minimum; otherwise the development of each subsystem would be limited by the pace of the work on the other subsystems. Furthermore, all the dependencies must be known: through a failure to list all the relations between the subsystems the project may result in a set of elements that appear to work individually but cannot be put together to produce a complete working system. This leads to failure to satisfy the overall requirements of the original problem.

A well known example of a method satisfying the decomposability criterion is called the “top-down” design [26]. Basically the method directs designers to start with a most abstract description of the system’s function, then refine this view through successive steps, decomposing each subsystem at each step into a small number of simpler subsystems until all remaining elements are of a sufficiently low level of abstraction to allow direct implementation. A typical counter example is a global initialization module; included in every software system produced. Many modules in a system will need some kind of initialization, such as opening certain files or initialization of certain variables, which the module must execute before it performs its first useful tasks. Although it may seem a good idea to concentrate all such actions, for all modules of the system, in a single module, to do so would endanger the autonomy of modules. Therefore the initialization module would need to have access to many separate data structures belonging to the various modules of the system and requiring specific initialization actions. This is incompatible with the decomposability criterion which states that every module will be responsible for the initialization of its own data structures [25, p. 41].

As it will be shown later, AOP is trying to overcome the issue of a global service, for example, where logging or database access is required and where all classes need to connect to this service in order for that system to work properly.

b) Modular composability

The Modular composability criterion is satisfied when a method favours the production of software elements which may then be freely combined with each other to produce new systems, possibly in an environment quite different from the one in which they were initially developed [25, p. 42]. Composability is the reverse process of decomposability; extracting software elements that are sufficiently autonomous from the context for which they were originally designed so that they may be used again in a different context. Composability is directly connected with the goal of reusability: the aim is to find ways to design software elements performing well-defined tasks and usable in widely different contexts.

Composability and decomposability are independent of each other and they don't match at all. The top-down design, for example, which is a technique that is favoured by decomposability, tends to produce modules that are not easy to combine with modules coming from other sources. This is because the method suggests developing each module to fulfil a specific requirement, corresponding to a sub-problem obtained at some point in the refinement process. Such modules tend to be closely linked to the immediate context that led to their development, and are unfit for adaptation to other contexts. Also, it is important to note that both composability and decomposability are part of the requirement for a modular method and reflect the inevitable mix of top-down and bottom-up reasoning.

c) Modular understandability

The Modular Understandability criterion is satisfied when a method helps to produce software in which the human reader can understand each module without having to know the others, or, at worst, by having to examine only few of the others [25, p. 43]. The importance of this criterion follows from the influence on the maintenance process. Most maintenance activities involve exploring existing software. A method cannot be called modular if a reader of the software is unable to understand its elements separately. This criterion, like the others, applies to the modules of a system description at any level: analysis, design implementation. The modular understandability criterion also affects the maintenance of the implementation and makes it harder to give the

implementation task to a team member as the implementation touches many segments that other team members are working on; an issue that AOP solves.

d) Modular continuity

The Modular continuity criterion is satisfied if a problem specification triggers a change of just one module, or a small number of modules in the software architecture that it yields [25, p. 44]. This criterion is directly connected to the general goal of extensibility. It is a known fact that “change” is an integral part of the software construction process. The requirements will almost inevitably change as the project progresses. Continuity means that small changes should affect individual modules in the structure of the system, rather than the structure itself.

e) Modular protection

In a similar manner the modular protection criterion is satisfied when the effect of an abnormal condition occurring at run time in a module remains confined to that module, or at worst only propagates to a few neighbouring modules in the software architectures that it yields [25, p. 45]. This criterion is for errors and failures within a software system such as run-time errors, resulting from hardware failures, erroneous input or exhaustion of needed resources (e.g. memory storage). It is important to mention that the method does not address the correction of errors, but the aspect that is directly relevant to modularity which is “propagation”. A good example of modular protection is the use of exception handling because is validating input at the source.

2.2.2 Rules for Modularization

Following the five fundamental requirements that should be satisfied for a modular design method, four rules are suggested to ensure the sustainability of “modularity”. The first rule addresses the connection between a software system and external systems. The rest address a common issue called “communication between modules” that is important for obtaining good modular architectures [25].

a) Direct Mapping

The modular structure devised in the process of building a software system should remain compatible with any modular structure devised in the process of modelling the problem domain [25, p. 47]. This means that when a good model is obtained from the problem domain it is desirable to maintain clear correspondence (mapping) between the structure of the solution and the structure of the problem. This rule follows from two of the modularity criteria:

- Continuity: Thus keeping a trace of the problem's modular structure in the solution's structure will make it easier to assess and limit the impact of changes.
- Decomposability: if some work has already been done to analyze the modular structure of the problem domain, it may provide a good starting point for the modular decomposition of the software.

b) Small and Explicit Interfaces

This Small Interface rule follows from the criteria of Continuity and Protection stating that if two modules communicate, they should exchange as little information as possible and must be public [25, pp. 48-50]. The Explicit Interface rule stands from the criteria of Decomposability and Composability (decompose a module into several sub-modules or compose it with other modules; any outside connection should be clearly visible).

c) Few interfaces

This rule follows in particular from the criteria of continuity and protection which states "if there are too many relations between modules, then the effect of a change or of an error may propagate to a large number of modules". Communication may occur between modules in variety of ways but with as few others as possible. [25, p. 47] Modules may call each other, share data structures etc. This rule limits the number of such connections. One way for this to be achieved is shown in Figure 1, "one node to all" is preferred to "maximised nodes" where each module is connected to all other modules.

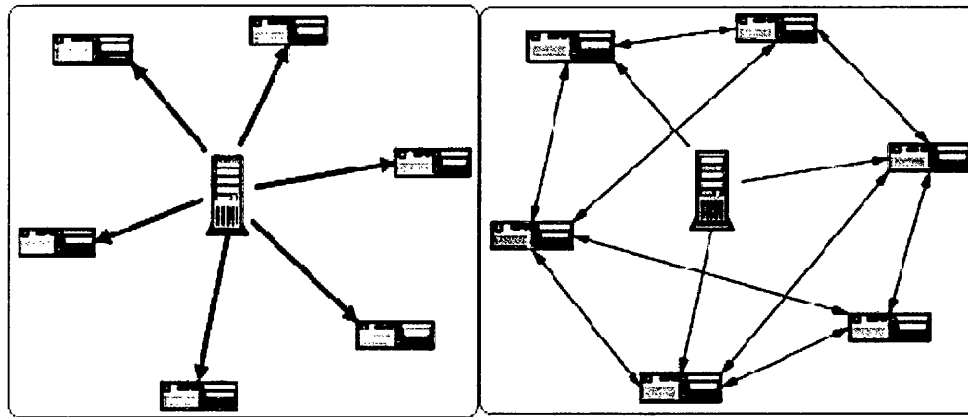


Figure 1 “one node to all” and “maximised nodes”

The one node to all is an extremely centralized structure where the master module communicates to all the other modules. The problem with this communication method is that if the master module fails the entire system would fail. Therefore, depending on the requirements other communication channel configurations can be used.

d) Information hiding

The designer of every module must select a subset of the module’s properties as the official information about the module, to be made available to authors of client modules [25, p. 51]. Application of this rule assumes that every module is known to the rest of the world through some official description or public properties. Obviously the whole text of the module itself (program text, design text) could serve as the description but this rule states that this should not be the case. The description should include some of the module’s properties; the rest should remain non-public or private. The fundamental reason behind this rule is the continuity criterion. Assume a module changes, but the changes apply only to its private elements leaving the public ones untouched; then the clients will not be affected. The smaller the public part, the higher the chances that changes to the module will be in the secret part. Imagine a module information hiding as an iceberg; only the tip (interface) is visible to the clients. Information hiding emphasizes separation of function from implementation. The key to information hiding is not management or marketing policies as to who may or may not access the source

text of a module, but strict language rules to define what access rights a module has to properties of its suppliers. [9]

2.3 Constraints of Object Oriented Technologies

An important assumption about software design processes and programming languages is that they exist in a mutually supporting relationship. Design processes break a system down into smaller and smaller units. Programming languages provide mechanisms that allow the programmer to define abstractions of system sub-units, and then compose those abstractions in different ways to produce the overall system. A design process and a programming language work well together when the programming language provides abstraction and composition mechanisms that cleanly support the kinds of units the design process breaks the system into. From this perspective, many existing programming languages, including object-oriented languages, procedural languages and functional languages, can be seen as having a common root in that their key abstraction and composition mechanisms are all rooted in some form of generalized procedure (GP) [21]. This doesn't ignore the OOP advantages it makes it simpler to focus on what is common across all GP languages. It was mentioned that the design methods that have evolved to work with GP languages tend to break systems down into units of behaviour or function. This style has been called by Parnas functional decomposition [23]. The nature of the decomposition differs between the language paradigms, but each unit is encapsulated in a procedure/function/object. In each case it is best to discuss it as a functional unit of the overall system.

When a programmer is writing an application there is some notion of “design” of the main features and functionalities that the application must support and how it might be represented in the code. The “ideal” mapping from design-level to source code implementation would be to have a simple and clear one-to-one correspondence i.e. each requirement would have a unique correspondence with an implementation construct. For example if the program needs to deal with an Employee, it would be ideal if the concept of the employee had a one-to-one mapping to an Employee class. The Employee class encapsulates everything the program needs to know about working with employees. If

there were different kinds of employees they could be mapped into an Employee class hierarchy. Therefore it is clear which portion of the implementation correspond to the design-level notions of the Employee.

A clear and simple one-to-one mapping from design level concepts to their source code implementation makes the application simpler to understand and maintain. The concepts and requirements at the design level correspond closely to the units of change over the program's lifetime i.e. if a new kind of employee is needed a new class can be added to the employee hierarchy. In the same way, if it is no longer required to keep a track of salary it can be deleted from the Employee class.

However not all design-level requirements are easy to have a clear one-to-one mapping with an implementation construct when using an object-oriented (OO) language. Consider, for example, the requirement that a view be notified whenever the state of an employee object it is displaying is updated. Usually this would be implemented by fragments of codes across the Employee hierarchy instead of an encapsulated module.

Below an extract of the code is shown:

```
public class Employee {  
    private String name;  
    private Double salary;  
    private Date birthDate;  
    private List listeners;  
  
    public Employee(...) {...} // details omitted  
  
    public void addListener (EmployeeListener listener) {  
        listeners.add(listener);  
    }  
  
    public void removeListener (EmployeeListener listener) {  
        listeners.remove(listener);  
    }  
  
    public Date getbirthDate () {  
        return this.birthDate;  
    }  
  
    public Double getsalary () {
```

```

        return this.salary;
    }

    public String setname (String employeeename) {
        this.name = employeeename;
        notifyListeners(this);
    }

    //etc.
}

```

The Employee class has methods to add and remove listeners, and has calls to a notifyListeners method every time the state is changing. Hence instead of a simple and nice one-to-one mapping, there is a one-to-n mapping known in AOP community as “scattering”. In general whenever a one-to-n mapping occurs from design-level concepts and requirements to implementation constructs the following problems can be expected: [27, p. xix]

- It is harder to understand and reason about the implementation of the requirement, because to get the full picture the developer needs to look in multiple places in the source code.
- It is harder to add or remove the implementation of the requirement from the code base. It is required to remember to add or remove logic at each relevant point.
- It is harder to maintain the implementation. As shown in the previous example any occurring changes must be consistent and correct across the application.
- It is harder to give the implementation task to a team member. The implementation touches many segments that other team members are working on.
- It is harder to reuse the implementation in another system. The implementation pieces are not modularized in a way that can be easily extracted and there are a lot of other dependencies from the current system tangled in with it.

When an application has multiple design concepts and requirements and some of them are one-to-n mappings, it inevitably ends up with source modules that contain logic to

do with multiple concepts and requirements. In the case of the Employee class it exhibits a two-to-one mapping ratio: one single module is implementing both the Employee concept and the “view notification” requirement. This is also known as “tangling” i.e. the different implementation components have been tangled together inside a single module [27, p. xx].

Therefore, failing to modularize crosscutting concerns leads to two things:

- 1) Code tangling (coupling of concerns)
- 2) Code scattering (the same concern spread across modules)

Figure 2 shows another example whereby a system consisting of a Bank, a Customer and Reporting Service has both code tangling and code scattering as it is evolving.

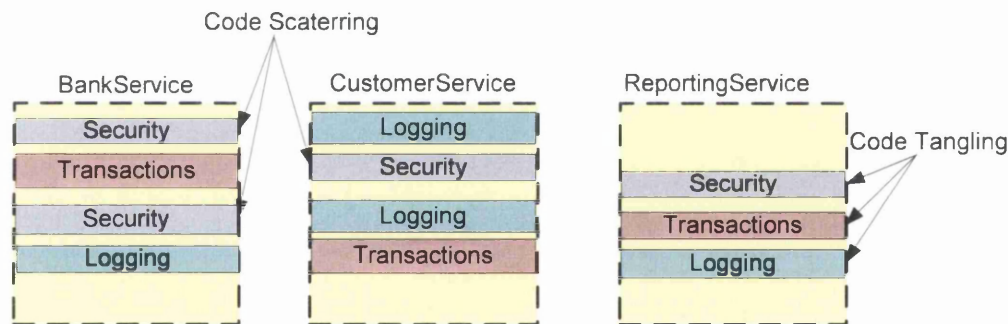


Figure 2 System evolution without modularization

On the other hand as shown in Figure 3, if the same system concerns were dealt with as aspects the system would achieve a better modularity as it would achieve a one-to-one mapping. Aspects are a unit of modularity, encapsulation and abstraction with the difference that aspects can be used to implement crosscutting concerns in a modular fashion. Aspects will be explained later in more detail.

Therefore when any application contains a one-to-n, n-to-one or n-to-n mapping between design-level concepts and requirements to implementation constructs it has strayed from the goal of simple, clear, direct one-to-one mapping. OOP does not provide the tools to cleanly map all concepts and requirements into a modular constructs whereas AOP is about getting as close as possible to a one-to-one mapping.

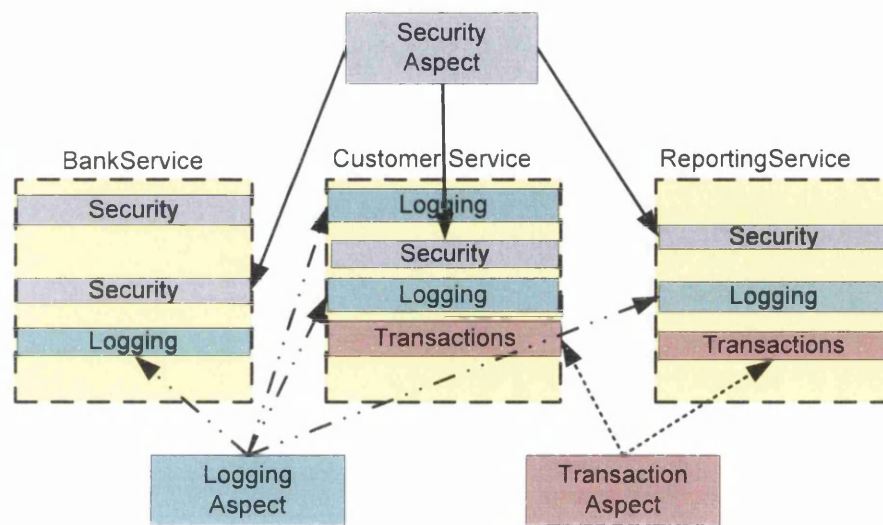


Figure 3 System evolution: AOP Based

2.4 Aspect-oriented Programming

AOP is a new evolution in the line of technology for separation of concerns which means technology that allows design and code to be structured to reflect the way developers want to think about the system [28, pp. 33-38]. AOP grew at the Palo Alto Research Center (PARC) during the 1980's and 1990's and the first paper to use the term was titled "Aspect-oriented Programming" and was published in June 1997 [21].

Kiczales et al. [21] state the reason and purpose of this programming technique. It explains that OOP was presented as a technology that can fundamentally aid software engineering, because the underlying object model provides a better fit with real domain problems. However, many programming problems were found that OOP techniques were not sufficient to clearly capture all the important design decisions the program must implement. Instead, it seems that there are some programming problems that fit neither the objected-oriented approach nor the procedural approach it replaces. This forces the implementation of those design decisions to be scattered throughout the code, resulting in "tangled" code that is excessively difficult to develop and maintain. Then it presents an analysis of why certain design decisions have been so difficult to clearly capture in the actual code. These decisions address aspects, and show that the reason

they have been hard to capture is because they cross-cut the system's basic functionality. The paper presents the basis for a new programming technique, called Aspect-oriented programming, which makes it possible to clearly express programs involving such aspects, including appropriate isolation, composition and reuse of the aspect code.

Aspect Orientation is not a completely new approach to writing software. For some time there have been many technologies that existed previous to AOP and now are placed under the banner of Aspect Orientation [29]. In the same way as virtual machine systems was not an entirely new concept when Java became recognized and adopted by the software community. The significant difference is in the philosophy behind the approach and how that philosophy drives the technology and tools. Hence, Aspect orientation is a new and more modular implementation of the advantages of the object orientation technologies [29, p. 1].

In objected-oriented analysis and design the requirements and statements are like nouns and verbs. Nouns become candidate classes and verbs become candidate methods of those classes. As discussed, AOP enriches OOP and other conventional paradigms by giving a new way to modularize the implementation of adverbs and adjectives. For example a thread-safe class or secure transaction. Adverbs and adjectives exist in order to define concepts independent of nouns and verbs to which they apply. Because they can be applied to many different entities they are a form of a crosscutting concern.

In the same way that for a design method to be called modular, fundamental design requirements need to be satisfied. There are various attempts to summarize AOP properties to satisfy the requirement of successful separation of concerns. Some of these suggestions were featured at the special edition for AOP at Communications of the ACM [28]. In brief, Mehmet Aksit summarizes the key issues of AOP properties using the following six "C"s:

1. Crosscutting is a behaviour that is used across the scope of a piece of software.

2. Canonality (i.e. conforming to well-established rules or patterns) is necessary for the stability of the implementation of concerns.
3. Composability is necessary for providing quality factors such as adaptability, reusability, and extensibility.
4. Computability is necessary for creating executable software systems.
5. Closure is necessary for maintaining the quality factors of the design at the implementation level.
6. Certifiability is necessary for evaluating and controlling the quality of design and implementation models.

And Harold Ossher [28] suggests also the four "S"s for successful separation of concerns. These are:

1. Simultaneous coexistence of different decompositions is very important.
2. Self-contained separation. Hence, each module should declare what it depends on, so that it can be understood in isolation.
3. Symmetric separation. They can be composed together most flexibly which means that there should be no distinction in form between the modules encapsulating different kinds of concerns. E.g. aspects are able to extend other aspects as well as classes.
4. Spontaneous separation that would make possible to identify and encapsulate new concerns, and even new kinds of concerns, as they arise during the software life cycle.

Therefore, AOP builds on existing technologies and provides additional mechanisms that make it possible to affect the implementation of systems in a crosscutting way. As mentioned crosscutting concern is a behaviour, and often data, that is used across the scope of a piece of software. It may be a constraint that is a characteristic of the application or a behaviour that every class must perform. In other words two concerns crosscut if the methods related to those concerns intersect. [29, p. 2] An example of a crosscutting concern was already shown earlier on with the Employee class and the

requirement that a view be notified whenever the state of an employee object it is displaying is updated.

Another classic example (also known as the “Hello world” example for crosscutting concerns) is one in which there are two concrete classes of Figure element, points, and lines [28]. These classes manifest good modularity, in that the source code in each class is closely related (cohesion) and each class has a clear and well-defined interface. But consider the concern that the screen manager should be notified whenever a Figure element moves. This requires every method that moves a Figure element to do the notification.

This is illustrated in Figure 4. Every method that must implement this concern is highlighted, just as the Point and Line boxes are drawn around every method that implements those concerns. It can be noticed that the box for DisplayUpdating fits neither inside of nor around the other boxes instead it cuts across the other boxes. Hence, is called a crosscutting concern. Using just OOP, the implementation of crosscutting concerns tends to be scattered out across the system, just as it would be here. Using the mechanisms of AOP, the implementation of DisplayUpdating behaviour can be modularized into a single aspect, which, can be seen as a single design unit. In this way Karl Lieberherr said that the programming language mechanisms of aspects can allow aspects to be thought even at the design level [28]. These aspects are also known as early aspects which are defined as crosscutting concerns in the early life cycle phases including the requirements gathering, requirements analysis, domain analysis and architecture design phases, i.e. early aspects refer to crosscutting properties at the requirements and architecture level. Examples of such properties include security, mobility, availability and real-time constraints [30], [31], [32] . Further discussion on crosscutting concerns in the early life cycle phases will be covered later.

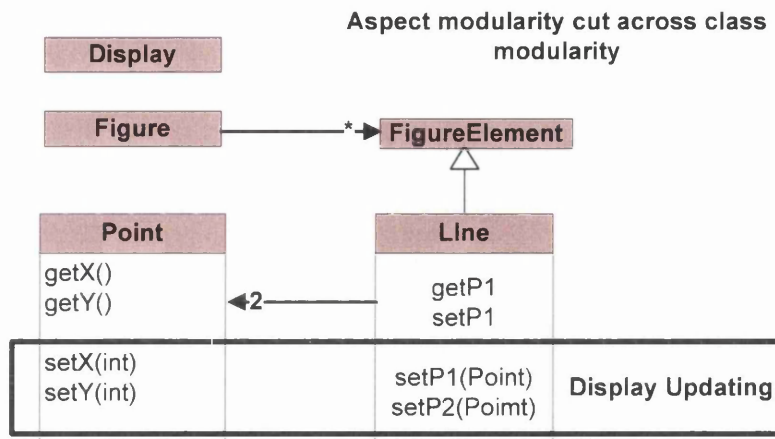


Figure 4 Aspects crosscut classes

In AOP, a single aspect can contribute to the implementation of a number of procedures, modules, or objects. The contribution can be homogeneous, for example by providing a logging behaviour that all the procedures in a certain interface should follow; or it can be heterogeneous, for example by implementing the two sides of a protocol between two different classes [28]. Like a class, an aspect is a unit of modularity, encapsulation and abstraction with the difference that aspects can be used to implement crosscutting concerns in a modular fashion.

A second key benefit that aspects provide is that they encapsulate the implementation of the feature or function that they implement. As already explained encapsulation means that all information relating to the implementation of the feature is hidden from other modules. Aspects also provide a powerful form of information hiding that classes cannot. This is done by being able to hide how and when something is done. For example, it would be hard to implement the requirement that any errors occurring within the control flow of an application due to user interaction should be flagged and all other errors logged without the use of aspects. This is because the information about the application for error handling would leak into all places that the error might occur. Aspect-oriented approach provides a set of semantics and syntactical constructs in order that aspects can be applied generically regardless of the type of software being written.

These constructs are advice, join points, and pointcuts. Advice is called the code that is executed when an aspect is invoked. Advice contains its own set of rules as to when it is invoked in relation to the join point that has been triggered. Join points are specific points within the application that may or may not invoke some advice. The specific set of available join points is dependent on the tools and the programming language being used under development. Pointcuts are a mechanism for declaring an interest in a join point to initiate a piece of advice. They encapsulate the decision-making logic that is evaluated to decide if a particular piece of advice should be invoked when a join point is encountered.

Another major key issue is the reusability of aspects. To make aspects more reusable “aspectual collaborations” concept can be introduced [31]. An aspectual collaboration describes an aspect using a class graph. When the collaboration is used, the class graph is mapped into a larger class graph using an adapter. Aspectual collaborations and adapters lead to better separation of crosscutting issues expressed in adapters and reusable behaviour expressed in aspectual collaborations. It is not good enough to modularize crosscutting concerns because the modularization might scatter another concern leading to a program that is still hard to maintain. It is therefore important to modularize crosscutting concerns such that they are loosely coupled to other parts of the program. The usefulness of reusability of aspects is covered in more detail later in the thesis.

Also, during early AOSD conferences [32], some papers argued [33, pp. 1-4] that the current AOP languages do not provide the third point of the benefits quoted by Parnas [9] i.e. comprehensibility, because they require systems to be studied in their entirety. Also in [34, p. 327] arguing for AOP, states that the modularity of a system should reflect the way developers would like to think about modularity, rather than the way in which developers are forced to think about it due to the language or other tools. Current aspect-oriented languages such AspectJ, however, do have tools and mechanisms that compensate this lack of modularity. Furthermore a preliminary evaluation has showed [33, p. 11] that with some modifications the language can provide sufficient flexibility

according to second criteria of Parnas. This discussion will be covered in more detail in the thesis.

2.5 Summary

This chapter starts with a survey in order to establish a perspective of the programming language evolution. Next, the concepts of modularization were introduced as a mechanism for improving the flexibility, efficiency, extensibility, reusability and comprehensibility of a system while allowing the shortening of its development time. The meaning of modularization and the benefits expected from modular programming are also explained. Criteria were suggested when decomposing a system into modules and discuss design requirements for modular methods. These requirements are decomposability, composability, understandability, continuity and protection. Furthermore, four rules were added to ensure the sustainability of modularity. These are direct mapping, fewer, smaller and explicit interfaces and information hiding.

Assumptions about software design processes and programming languages were discussed and it was shown that a design process and a programming language work well together when the programming language provides abstraction and composition. These mechanisms can cleanly support the kinds of units the design process breaks the system into and a clear and simple one-to-one mapping from design level concepts to their source code implementation. This helps the application simpler to understand, easier to maintain and reuse it in another system. It was also shown how some of the concepts of modularity are hard to capture in the conventional object oriented programming and how AOP offers a clear and simple one-to-one mapping from design level concepts to their source code implementation which also helps the program to be simpler to understand and maintain. These are known as Aspects and they provide a mechanism by which a crosscutting concern can be specified in a modular way. Aspect-oriented approach provides a set of semantics and syntactical constructs in order that aspects can be applied generically regardless of the type of software being written. These constructs are advice, join points, and pointcuts. Finally it was suggested the

importance to modularize crosscutting concerns such that they are loosely coupled to other parts of the program.

3. AOP Language Metamodel

3.1 Overview

The previous chapter presented modularization as a mechanism for improving a system in terms of management, product flexibility and comprehensibility [35]. It was shown also that there are design decisions that a system must implement in a modular fashion but are difficult to express and define them clearly because they crosscut the systems' functionality [36]. It was mentioned earlier that this research attempts to show the way that AOP provides support for these design decisions. The first contribution towards this goal is captured in this chapter.

This chapter aims to reflect and analyse the state-of-the-art in AOP techniques that would provide the tools to assess and compare AOP versus other programming approaches. The first step towards this aim is to survey AOP technologies and frameworks and investigate language models and meta-models for AOP. This would allow a more general but comprehensive comparison and analysis of the fundamental aspect language features as well as their implementation and execution techniques.

When searching for AOP languages or frameworks issues may arise due to the uniqueness of each of the tools because not all have been developed equally and for the same purpose and due to the open source nature of many AOP projects many have contributed either out of interest or trying to resolve some of the problems they encountered in their research or projects. Furthermore, although the principles of programming maybe the same but the development or approach of the project varies which, makes standardized information difficult to obtain.

When AOP started to gain momentum and was featured as the major themes in many journals such as communications of the ACM [37, pp. 28-32] many research groups and developers started to classify AOP languages and frameworks in different ways. For example [38] suggested that AOP should be classified based on their implementation approaches. These categories were defined as (1) class-weaving-based (bytecode) and

(2) proxy-based. Typical examples of the first approach are AspectJ [39] and JBOSS [40] where the crosscutting concerns are implemented independently and the weaving can be performed at compile, load and run time. Examples of the latter approach are SpringSource [41], Nanning [42] where the method invocations on an object can be intercepted to inject custom code and they typically use JDK dynamic proxy [43], CGLIB proxy [44], or both.

For this reason the Aspect-Oriented Software Development (AOSD) community started a research language lab [18]. AOSD community started soon after the first time the term AOP was

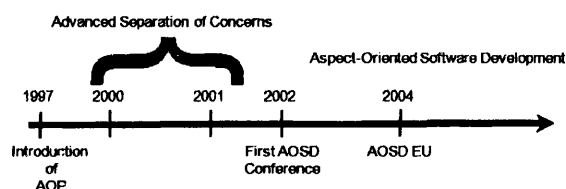


Figure 5 AOSD Timeline [32]

published (June 1997) and held its 1st International Conference in April 2002 in Netherlands [45]. The purpose of the conference was to create a forum for dissemination and discussion of leading-edge research and for researchers in the field to get together. In addition the opportunity was given for practitioners to learn about AOSD technologies, the practical advantages they offer and meet with the inventors and providers of those technologies. In similar fashion the opportunity for researchers to learn from practitioners about real-world technical problems that can motivate further research, discuss the challenges faced when adopting AOSD in industry and what can be done to address them [45]. Figure 5 shows AOSD timeline [45].

AOSD became an emerging paradigm that provided explicit abstractions for concerns that tend to crosscut multiple system components and result in tangling in individual components [46]. It started at the programming level of the software development life-cycle and in the last decade several AOP languages were introduced such as AspectJ [39], HyperJ [47], ComposeJ [48], DemeterJ [49] etc. As the number of activities, languages and innovations increased the need of a unified network was more immanent. The European Network of Excellence on AOSD emerged to harmonise and integrate the research, train and disseminate the activities of its members in order to address fragmentation of AOSD activities in Europe and strengthen innovation in areas such as

aspect-oriented analysis and design, formal methods, languages and applications of AOSD techniques in ambient computing. The European Network of Excellence also acts as an interface and a centralised source of information for other national and international research groups, industrial organisations and governmental bodies to access the members' work and enter collaborative initiatives [18].

The AOSD-Europe project structures its research labs in five areas [18]:

1. Analysis and Design Lab focuses on requirements engineering, architecture and design research.
2. Languages Lab focuses on research in language models, meta-models and language implementation.
3. Formal Methods Lab focuses on formal specification and verification research.
4. Applications Lab focuses on key concerns needing AOSD, adaptive AO middleware and demonstrator applications.
5. Atelier provides the integration dimension for the labs in terms of a development methodology, language implementation toolkit and a framework for IDE integration.

Therefore, it deemed appropriate to start the investigation of AOP languages from the AOSD languages lab where it adapts aspect-oriented languages that are concrete, high-quality with a clean design, supported by advanced implementation technology and preferably with production support and quality. The lab's main goals are design space and implementation and runtime support technology:

Design space:

- a. Identification and description for aspect-oriented languages that all partners agree.
- b. The advancement of language constructs for each of the points identified.
- c. The integration and cooperation along a common theme of interest.

Implementation and runtime support technology:

- a. The advancement of current language implementation processes
- b. To increase direct support of the specifics of aspect-oriented language concepts

Consequently, the purpose of these goals is to investigate language models and meta-models for aspect-oriented programming as well as an inventory of aspect language implementation platforms and techniques [18].

3.2 Language Models

As already mentioned, in order to achieve the aim of this research a survey of AOP languages was to be conducted which could enable to define a common model for comparison and analysis purposes. However, the AOSD Languages Lab had already performed an extensive survey on twenty seven AOP languages according to particular dimensions of interest ensuring that each language is appropriately reviewed and the commonalities and the variations of each language identified. This is very important as it can be used as an input on the classification of aspect languages and a common metamodel. The survey consisted of two different categories the first is the language model where the focus is the language itself and the latter is the execution model where the focus is on the implementation of the woven code i.e. the output of the aspect weaver. It is worth mentioning that in the survey not all aspect languages are represented in both categories. This selection was determined based on initial interest by all partners, on available information about the languages and the observables differences. Furthermore, many language implementations only have a proof-of concept execution model, which are not very interesting from the survey's point-of-view [50].

As with the survey, the AOSD Languages Lab had already defined an initial language metamodel for AOP languages representing a fundamental characterization of their essential language features. An intermediate step towards this metamodel was refining the survey into a taxonomy of aspect languages which, helped to identify some of the major properties in each dimension of interest. Another important dimension in the design space is investigating join point models and pointcut languages [18].

In terms of the aspect-oriented execution models as already mentioned the focus of the languages lab is on the description and comparison of implementation and execution mechanisms for aspect-oriented language features. The survey analysed more than 17

different AOP tools on several platforms and implementation languages, from Java over .NET to C and Smalltalk. All approaches were analysed according to a common structure, so that the descriptions would contain information at the same level of detail for all surveyed tools. The results led to the formulation of an inventory of aspect-oriented execution models presenting technical documentation about implementation approaches for AOP execution models such as the representation of AOP entities in an execution model, the implementation of an execution model's join point and pointcut models, a model's approach to weaving, its way of managing advice instances, and support for distribution. For each of these mechanisms, the design space has been analysed and the various ways of implementing the mechanism have been documented [18]. The list of the languages that the survey covers can be found in the Appendix. Note that this research will not discuss the execution side of the language model.

3.3 Survey Dimensions and Results

Each language and execution model in the survey has to be described among the same dimensions of interest. AOSD Languages Lab defined a set of questions regarding what the dimensions should be in agreement with all language lab partners [50, p. 14]. Figure 6 depicts the set of dimensions that were agreed and includes the related questions that define each aspect language dimension. The execution model dimensions were also defined and can be found in the Appendix.

Figure 2 illustrates the six dimensions of interest that describe the languages conducted in the survey [50, p. 14]. In the taxonomy of aspect languages [51], which was derived from this survey, the major commonalities and variations between the surveyed aspect languages were filtered and had an impact in the dimensions of interest in order to reflect better the essential dimensions.

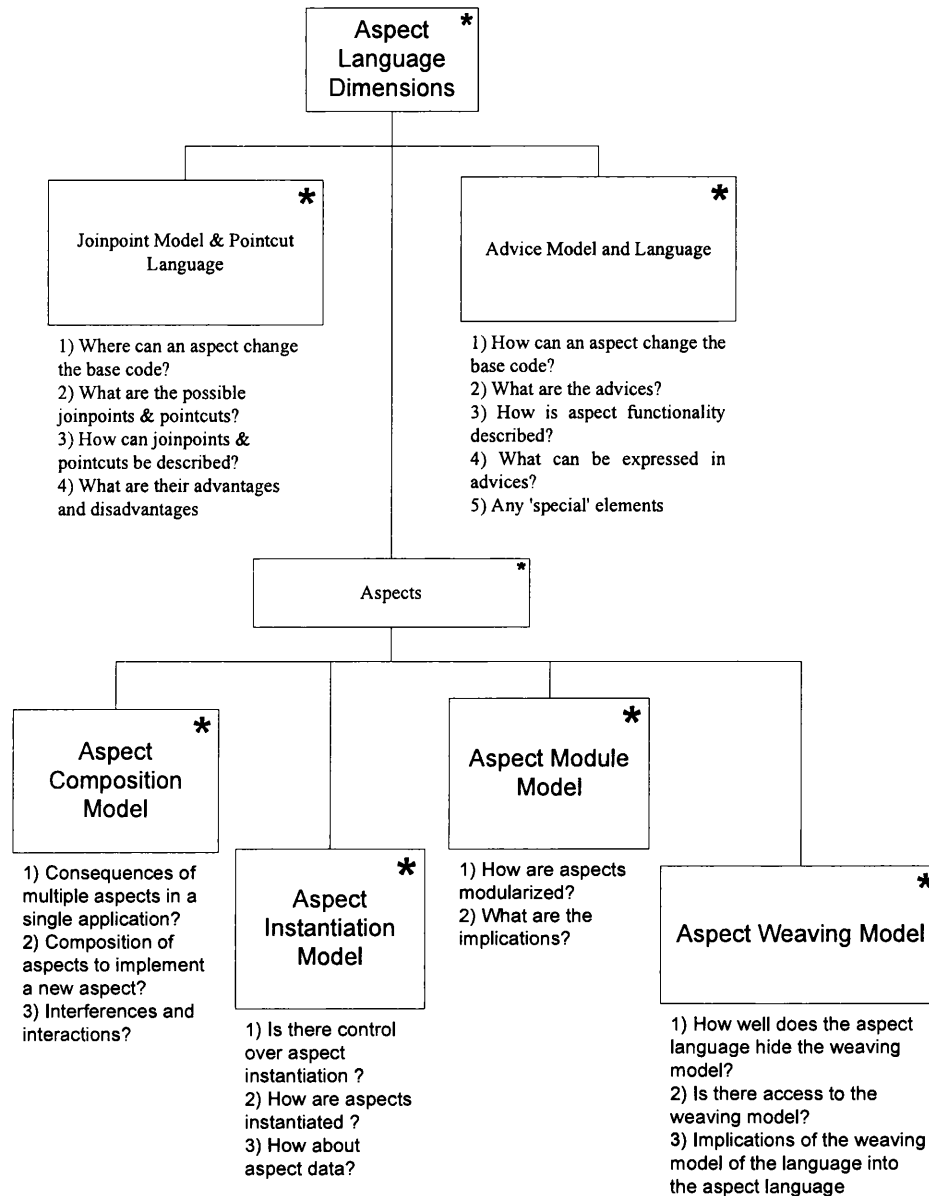


Figure 6 Aspect Language Dimensions

This meant that the module and composition models were merged; the join point model and the pointcut language separated and the weaving model discontinued as it relates to a specific implementation of certain aspect languages and does not reflect the essential concepts of an aspect language. Therefore, the new view of the dimensions is the join point model, pointcut language, advice model and language, aspect module and composition model and aspect instantiation model.

The next step is to investigate language models in order to create meta-models. The conception of a metamodel for AOP languages can give a fundamental understanding of what can be done. Similar to how OOP languages can be characterized by concepts such as object identity, encapsulation and polymorphism [52] this metamodel describe an initial characterization of AOP languages. The metamodel entailing a common understanding of AOP languages will allow collaboration and integration activities between the designers of these languages. Furthermore, these activities need to be supported by an experimental environment such as the language implementation toolkit of the Atelier (WP2) [53].

The Atelier, which means literally a studio especially for an artist or designer, is the activity leading the integration of the various tools, methods and techniques developed in AOSD-Europe, to create a "software workbench" for AOSD practitioners and researchers. In focussing on the creation of a "software workbench" the Atelier expects to act as a vehicle for technology transfer and to help to improve integration between activities within AOSD-Europe. The Language Implementation Toolkit (LIT) provides tools for building AOP language implementations; e.g. parsers, weavers, run-time environments, etc. The use of this toolkit provides the possibility to compare and integrate the different language features without focusing on implementation and performance details. The analysis of the surveyed aspect languages is an important step in the design of the metamodel because it results in an understanding of the fundamental commonalities and the important variability between aspect languages [54, p. 5].

Taking into consideration the questions of the dimensions shown in Figure 6 and the impact in the dimensions of interest after taxonomy the following common language features of aspect languages have been identified. These are join point model, pointcut language, advice model and language, aspect module and composition model and aspect instantiation model.

I. Join point model

Most AOP languages have a join point model for aspects to specify “when” they want control. When applications execute, methods are called, objects get initialized, fields are accessed and updated and constructors are executed. The join point model defines these events known as join points which are visible to an aspect when a program is running. The aspects specify or filter which of these events they are interested through a pointcut [27, p. 137]. The results of the taxonomy showed that most of the aspect languages have a dynamic join point model which means that the join points are points that can be directly identified in the execution of the program (static, event-based and state-based join point models are less common in the set of surveyed aspect languages). Also, an important number of aspect languages provided paradigm- or domain-specific join points. A domain-specific aspect language is used to express a concern that cuts across multiple concerns [55].

II. Pointcut language

A pointcut is used to select join points. It acts like a filter, matching join points that meet its specification and blocking all others. For example AspectJ supports three different categories of pointcuts. The first and most fundamental are join points based on the “kind” of join point i.e. the execution of an exception handler, the static initialization of a class. The second category matches join points based on “scope” i.e. checking is the join point has occurred within the control flow of a given operation. The final category matches join points based on “context information” at the join point itself i.e. checks whether the currently executing object is an instance of a given type [27, p. 139].

The results of the taxonomy showed that most of the aspect languages used pointcuts that were either (1) Query languages: a complete query language to match join points in the join point’s space i.e. contains all possible join points (primitive predefined predicates that can be combined into new user-defined predicates) or (2) Assembly of predicates: a limited version of a query language where pointcuts can only be created by grouping existing, pre-defined predicates.

In order to match a pointcut, most aspect languages offer predicates that can extract structural as well as behavioral properties from join points [54, p. 6].

III. Advice model and language

Advice contains its own set of rules as to when it is to be invoked in relation to the join point that has been triggered. As mentioned pointcuts are predicates that match join points, and advice specifies what to do at those join points that the pointcut matches. Each segment of advice is associated with a named or anonymous pointcut and specifies the behaviour that it wants to execute before, after or around, the join point that pointcut matches. Unlike method calls in which parameters values are explicitly passed by the caller, an advice declaration may contain parameters whose values can be referenced in the body of the advice and the parameter values are provided by the pointcut [27, p. 140].

The results of the taxonomy showed that all but a few aspect languages use the base language to express their advice and this is often an object-oriented language. The application of advices is almost always before, after and around constructions. Finally, most aspect languages offer join point reflection in the advice [54, p. 6]. This is very useful because join point reflection can be used to handle specific cases within a piece of advice when its pointcut matches several join points of different types or with different types of arguments. Reflection can also provide more information about a join point via the signature of the join point. The signature contains details about the point in the base code corresponding to the join point [56].

IV. Aspect module and composition model

The results of the survey showed that the majority of aspect languages offer an asymmetric aspect module concept. This means that the crosscutting concerns are modularized using a separate programming construct for aspects, which differs from the modules used to encapsulate the implementation of other concerns rather than modularizing all the concerns in the same kind of module. Since most aspect

languages represent aspects as a kind of classes, the object-oriented principles of specialization and substitutability [57] can often be applied to aspects [54, p. 6].

V. Aspect instantiation model

An aspect instance defines the values of the variables defined in an aspect and used in its advices. It seems that there is no general principle for aspect instantiation that is accepted by the vast majority of aspect languages. The paper by [58] discusses the shortcomings of AOSD languages, arguing that the lack of polymorphism and the difficulty with which aspect instances can be accessed and used within AspectJ, forces programmers to resort to less elegant solutions. For example by introducing code tangling in advice definitions, increasing code complexity and diminishing maintainability and robustness. This issue was addressed by [59] in the 2nd AOSD conference and they suggested potential solutions to this argument such as aspectual polymorphism as it makes aspects in any comparable AOSD language more expressive and reusable across programs, while preserving safety.

From the results of the taxonomy the aspect instantiation model is characterized with two distinct properties namely its specification and policy. The specification consists of explicit and implicit instantiation. In explicit instantiation the aspect state is only instantiated when the developer explicitly instantiates an aspect (i.e. sending a message to an aspect that creates an instance of that aspect). In implicit instantiation the aspect state is instantiated implicitly, which means the first time an aspect gets executed in a certain context, the state is initialized and that the aspect invocation mechanism selects the correct state for the aspect. In terms of its policy there are three possibilities: the first is when a single aspect definition (singleton) is associated with a single state and therefore there are no multiple states, the second is when the scope of the state (fixed scopes) can be determined by the developer but the possibilities are fixed by the language (there can be multiple aspect states for a single aspect definition) and thirdly the scope of the state (customizable scopes) can be determined completely by the developer (there can be multiple aspect states for a single aspect definition).

The analysis of the survey and taxonomy provides an understanding of the fundamental commonalities and the important variations between AOP languages. This understanding will help to define an initial metamodel as the fundamental characterization of the essential and diverse concepts present in the current aspect languages.

3.4 Common Language Concepts Metamodel

The construction of a collection of concepts within a certain domain, i.e. a metamodel, has been conceived by the AOSD Languages Lab as an open and extensible framework that makes it possible to describe and categorize aspect languages according to common language concepts and their semantics. These concepts represent essential aspect language features and according to their particular dimensions of interest four sub-metamodels have been defined: *the join point, pointcut, aspect binding and advice* metamodels that together are known as *the common language concepts framework metamodel* (common metamodel) [54, p. 10].

Any aspect language needs to be defined as a mapping of its own language features to the concepts in the metamodel. Hence, a framework approach has been taken in order to avoid oversimplification as specific language features of particular aspect languages can only be partially described as specializations of the concepts described in the common metamodel. Furthermore it gives the opportunity for all aspect languages to be described with respect to the framework metamodel instead of a separate metamodel for each aspect language. Therefore the framework approach is essential because it allows the users to describe specific features of aspect languages as specializations of the framework.

Figure 7 illustrates how the Aspect Language dimensions that were derived from the survey and the resulting taxonomy of aspect languages features feed in the creation of an initial metamodel that is an open and extensible framework. The aspect language concepts are defined as specializations of the concepts in the common metamodel and can also introduce new language concepts which are specific to one language and relate them to the concepts represented in the common metamodel. The framework is

complemented with an interpreter that describes the semantics of the common language concepts. The interpreter interprets instantiations of the model which is a fundamental part of the common metamodel as it implements the operational semantics of all language concepts [54, p. 8].

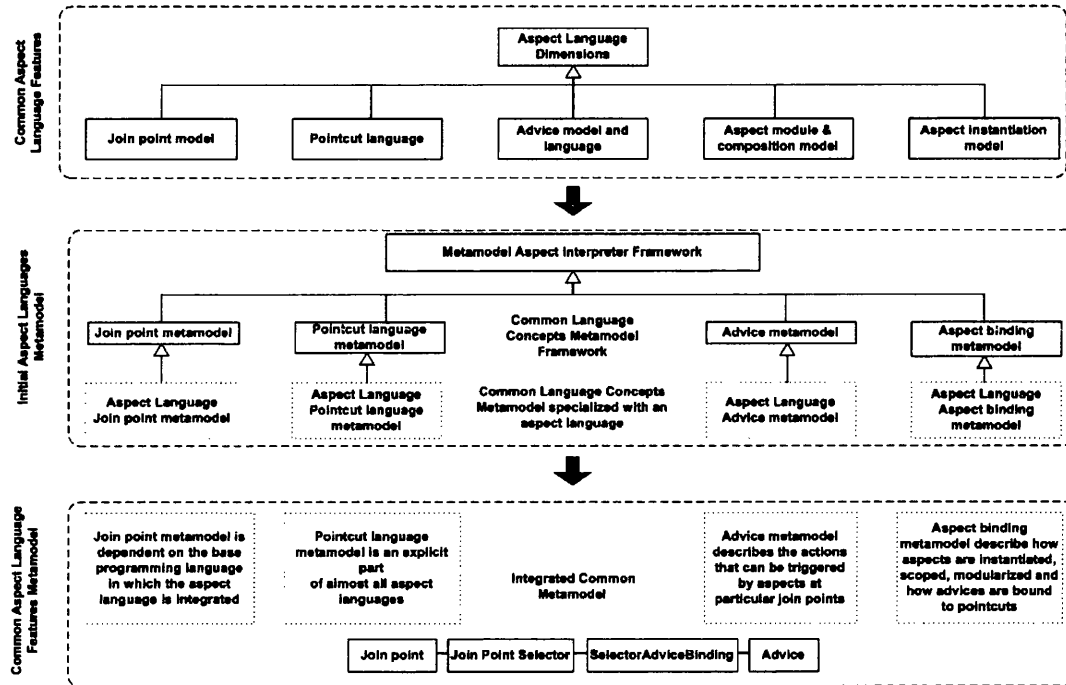


Figure 7 Survey Dimensions and Common Metamodel adapted from [54]

As stated earlier and shown in Figure 2 the model consists of four essential parts in the model where each part describes one or more important dimensions of an aspect language. These four parts also known as common language concepts metamodel will be explained in more detail in the following sections.

I. The Join Point Metamodel

The concept of the join point is the same as in the aspect language. The most widely used base languages for aspect orientation are object-oriented languages and this model is very much dependent on the base programming language in which the aspect language is integrated. Join points are essential in the execution of an application as they specify when aspects want control. The metamodel consists of

the structural part which refers to a location in the source code and the behavioural part that is a representation of the application's execution state.

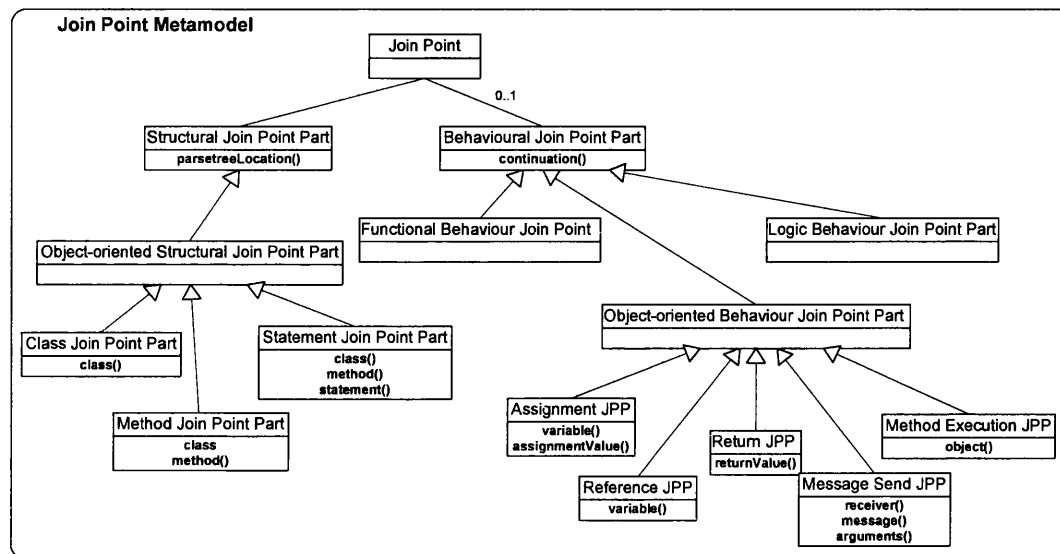


Figure 8 The Join Point Metamodel adapted from [54]

All join points (static or dynamic) are represented as dynamic join point in the metamodel. A dynamic join point consists of structural or behavioural part whereas the static join point has only structural part. Because most aspect languages have an object-oriented language as the base language the focus of the metamodel is on the object-oriented structural and behavioural elements. The general concept of a join point is covered in the metamodel as a point in the execution of a program but needs further specialization to reflect the different kinds of join points available in different aspect languages. The model that is illustrated in Figure 4 deals with join points in the execution of an advice because advices are executed in the same way as any other expression in the program that result the creation of the join points during the execution of advice (D39 – Language Lab, 2006, p. 10). Later in this chapter the metamodel illustrated in Figure 8 will be used to model an aspect language that is not covered in the AOSD survey.

II. The Pointcut Language Metamodel

Pointcut expressions are represented as predicates over join points i.e. they evaluate join points. If the join point is matched by the pointcut expression the evaluation returns true and false if it does not. Due to the existence of diverse pointcut languages various evaluators need to be represented. The language of a pointcut is a property of the join point selector and contains enumeration and query languages as well as reflection protocols present in the base language. Furthermore, the metamodel express the concept of the pointcut as a join point selector. This can be a primitive selector (single predicate to the join point) or composed selector (multiple predicates to the join point).

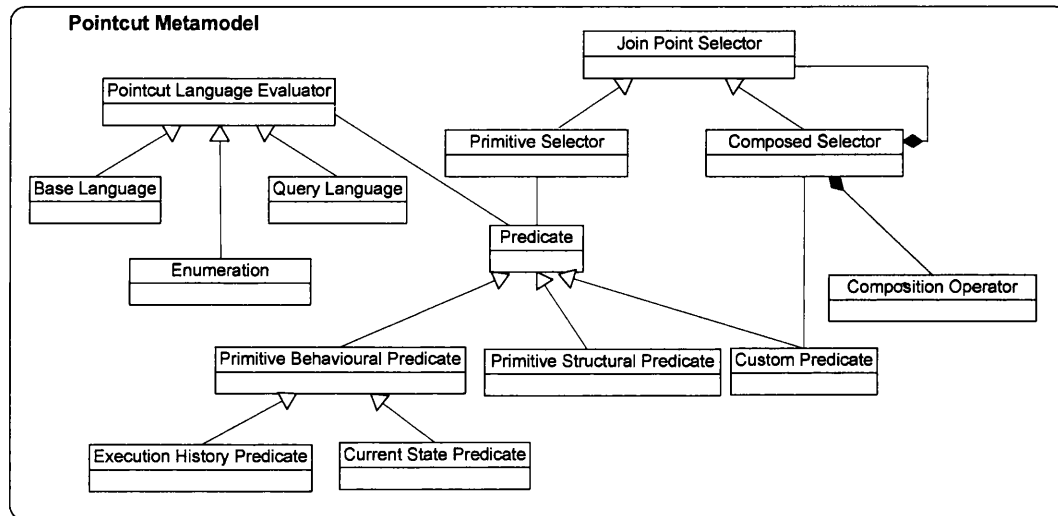


Figure 9 The Pointcut metamodel adapted from [54]

Also the join point metamodel consists of different kinds of predicates that can be applied to a join point in a selector. As shown in Figure 9 [54] these are *behavioural predicates* which deals with the behavioural properties of the join point. Behavioural predicates may be further specialized into *execution history* and *current state predicates*, the *structural predicate* which deals with the structural properties of the join point and finally the *composed predicate* which is a user-defined predicate that is expressed as a composition of selectors to be executed using operators. The *composed predicate* is defined as a set of selectors that each

applies another predicate to the join point and that are composed using operators (D39 – Language Lab, 2006, p. 12).

III. The Advice Metamodel

The *initial advice metamodel* use the same language as the base language but this is not a restriction of the entire metamodel. Following the selection of a join point, the advice metamodel describes the particular actions that can occur as the result of the application of an aspect. Advice, which express the functionality which needs to be invoked by an aspect, are modelled using strong advice actions that are composed as a tree structure. This structure, like the previous metamodels can be composed of primitive or composed base level i.e. normal application expressions and metalevel actions i.e. specific actions that can only be contained in an advice. Metalevel actions are explained in more detail when discussing metalevel operations. Furthermore, each advice action is related to the evaluator that needs to be executed hence different evaluators metalevel actions need to be defined. More details can be found in [54, p. 14].

IV. The Advice Binding Metamodel

The *aspect binding metamodel* represents aspects that consist of pointcuts, advice and variable declarations. An aspect has the *selectoradvicebindings* which relate to join point selectors, and advice definitions. Furthermore, each aspect also contains variable declarations. These define the state of an aspect 'instance'. A *stateselector* is associated with each variable in an aspect. The *stateselector* defines how a particular state is selected. Finally, a *bindingsselector* represents the composition of advices when multiple aspects and/or advices apply at the same join point. More details can be found in [54, p. 14].

The above metamodels are related to each other and integrated into the common language concepts metamodel. This is done as shown in Figure 3 by having the join point evaluated by the join point selectors which in turn, are bound to advice by a SelectorAdviceBinding.

3.5 Execution Semantics of the Metamodel Interpreter

A definition of a programming language is usually defined through semantics. Semantics is concerned with the interpretation or understanding of applications and how to predict the outcome of program execution. The semantics of a programming language describe the relation between the syntax and the model of computation [60]. There are several widely used techniques for the description of the semantics of programming languages also known as syntax-directed semantics. These are: [61]

- I. Algebraic semantics which describe the meaning of a program defining them in algebraic relationships and operations.
- II. Axiomatic semantics which define the meaning of the program implicitly. It makes assertions about relationships that hold at each point in the execution of the program.
- III. Denotational semantics which describe what is computed by giving a mathematical object such as a function which is the meaning of the program.
- IV. Operational semantics which define how a computation is performed by defining how to simulate the execution of the program. Operational semantics may describe the syntactic transformations which mimic the execution of the program on an abstract machine or define a translation of the program into recursive functions.
- V. Translation semantics which describe how to translate a program into another language usually the language of a machine. Translation semantics are used in compilers.

A language can also be defined by an interpreter [54, p. 15]. The description of the semantics of the metamodel can be done by using the implementation of an interpreter because the set of evaluation functions defined by the interpreter can have a close relation with its description using operational semantics. This can be seen as a first step towards formal semantics i.e. the field concerned with the rigorous mathematical study of the meaning of programming languages and models of computation [61].

Furthermore, the interpreter can provide executable semantics which establishes a solid ground for tools to investigate and experiment with the semantics of language features.

The following sections describe the concepts an interpreter employs to explain the semantics of the metamodel. These are: the base and metalevel aspect interpreter, discrete evaluation through join point stepping, continuations, woven execution of applications, metalevel operations, metalevel aspect state, and aspect environment.

I. Base and Metalevel Aspect Interpreter

The interpreter is separated into two parts: the base and metalevel aspect interpreter. Thus, when the interpreter evaluates an aspect-oriented application, the application entities can be expressed according their base or aspect-oriented language concepts. In order that the metamodel and its interpreter focus only on the aspect-oriented language concepts; the metalevel aspect interpreter evaluates aspect applications that are expressed using concepts of the metamodel and therefore the semantics of the aspect-oriented language concepts are localized in the definition and implementation of the metalevel aspect interpreter. However, this does not assume a clean separation of aspect and base languages at the language level

Also because aspects impose a different behaviour on the base program, an integrated behaviour of the base and aspect programs is required. This can be achieved when the metalevel aspect interpreter that interprets the aspect-oriented part of the program in a metamodel representation, controls the execution of the base interpreter which, interprets the base program part (shown in Figure 10). As a result, the execution of the aspect program essentially modifies the execution of the base program [54, p. 15].

II. Discrete Evaluation through Join Point Stepping

During the evaluation of a program, after every discrete evaluation step the base interpreter communicates join points to the metalevel aspect interpreter. After each evaluation step, the base language interpreter stops the execution of the program at

hand, creates a join point that represents the current execution state and passes control to the metalevel aspect interpreter. The aspect interpreter can then decide to invoke an aspect at this join point or it can decide to let the base interpreter continue its normal evaluation. These discrete evaluation steps are similar to the notion of continuation marks described in [62] as a mechanism for implementing an algebraic stepper. The stepper inserts a break point between each evaluation step to show the execution of a program. At each break point, the stepper prints representations of both the current value and the current continuation. Figure 6 illustrates these join points.

III. Continuations

The most essential concept to model the execution semantics of aspect languages is the notion of a continuation [54, p. 16]. The term continuation refers to an abstract representation of the control state. In other words it is questioning where in the application, which function and which line are being executed. Current continuation or continuation of the computation step is the instructions that will be executed after the current line of code is executed. In other words, it captures the current execution state of the program such that it can be stored and reconstructed later on. Hence, applications must allocate space in memory for the variables its functions use (call stack) because it allows for fast and simple allocating and automatic de-allocation of memory (heap) [63].

In the case of the metalevel aspect interpreter, it manipulates continuations of the base interpreter's program to model the semantics of the execution of aspect-oriented applications. When a join point triggers the execution of an aspect's advice, a continuation of the current base program is stored and a new continuation is created that executes the aspect's advice [54, p. 18].

IV. Woven execution of applications

The standard semantics of woven execution go through the suspension and re-activation of continuations. It is preferred to have the execution of the instruction at

the join point controlled through a metalevel action rather than omitting the execution of the instruction at the join point when re-activating a continuation [54, p. 19]. For example Common Aspect Semantics Base (CASB) framework defines the semantics of base and woven applications using the models of the execution semantics [64]. CASB is one of the main tasks of the formal labs. It aims to provide a framework with precise formal definitions of concepts and terminology of AOSD in order to prove the correctness of aspect transformations [65]. It allows the developer to inspect the woven program or to debug its execution in order to understand its semantics.

Furthermore, besides explaining briefly that the standard interwoven execution of applications goes through switching, suspension and activation of continuations it is important to mention there are some specific execution scenario's where the generality of the approach is illustrated by dealing with some aspect interaction scenarios [54, p. 21].

V. Metalevel Operations

It was explained earlier that the execution of an aspect-oriented program is the execution of a set of continuations, but then, how can the semantics of particular AOP language determine the way that an aspect-oriented program modifies these continuations and their execution? In addition, how can advices that contain specific expressions which cannot be understood at the base level be modelled using metalevel operations?

As shown in the advice metamodel, the metalevel operations are embedded in the advice and these metalevel operations are executed but not understood by the base interpreter. Therefore, the base interpreter's execution must be halted in order to execute the metalevel operations by the aspect interpreter. Through the survey [50] that was conducted three metalevel operations emerged. These are: continuation manipulation operations, aspect program operations and reification operations [54, p. 21].

a) Continuation manipulation operations

These operations manipulate the stack of aspect continuations. Every continuation keeps a list of applications that it already activated. This list is copied to the continuation that is created to execute the join point instructions. The list is emptied when a continuation is reactivated. Therefore, unless a continuation is restarted, it will never cause the activation of the same program at the same execution state [54, pp. 19-20].

b) Aspect program operations

These operations are necessary to model dynamic selector-advice binding semantics that can activate or deactivate aspects and allows to model aspect deployment and dynamic aspects. The field of metalevel actions that manipulate different parts of the aspect program is still developing and thus cause changes in the classification of aspect languages which could refine the metamodel [54, p. 20].

c) Reification operators

Reification is used when making a data model for a previously abstract concept. In this case, operators reify metalevel aspect values, such as join points, to the base level. The metamodel includes these metavalues and maps them onto the values used in the metamodel [54, p. 20].

VI. Metalevel Aspect State

An important aspect of the interpreter is keeping track of specific data relating to the execution of the aspect program. The specific data is metadata that consists of the execution of the base program and can be used by the aspect program in order to select join points, advice etc. Probably the most important part of the metalevel aspect state is the trace of all events, known as execution history, that happened in the base program since the evaluation started. This execution history helps to model pointcut predicates that reason about the state of the base program at some point in

time before the current state i.e. runtime stack of the base program. Examples of such predicates are found in stateful pointcuts or event-based pointcut languages [54, p. 20].

VII. Aspect Environment

Besides crosscutting behaviour an important factor of the metalevel operations is to consider the crosscutting state. By this is meant that subsequent advice activations of the same aspect may need to occur in the same scope and the variable initialization needs to happen when the advice needs to execute in a new scope. For this purpose metalevel aspect interpreter has a heap where references can be kept to actual variable values in the base interpreter's heap as shown in Figure 10. These references need to be kept because the advices are anyway executed by the base interpreter, which means that the variable values also need to be base language values [54, p. 21].

A summary of the concepts that interpreter employs to explain the semantics of the metamodel can be shown in Figure 6. It represents the base interpreter's runtime stack as a stack of frames (Fr) and the aspect interpreter's runtime stack as a stack of continuations also known as suspended continuations (Ct). For each such program that is executed by the base interpreter, a continuation is created which means that continuations are used to represent and store the state of the execution of the base program in the metalevel aspect interpreter. In a nutshell, each time the base interpreter halts the execution of the program at a join point; it passes this join point to the metalevel aspect interpreter. When an aspect needs to be invoked at this join point, the metalevel aspect interpreter stores a continuation that represents the execution state of the currently executing program.

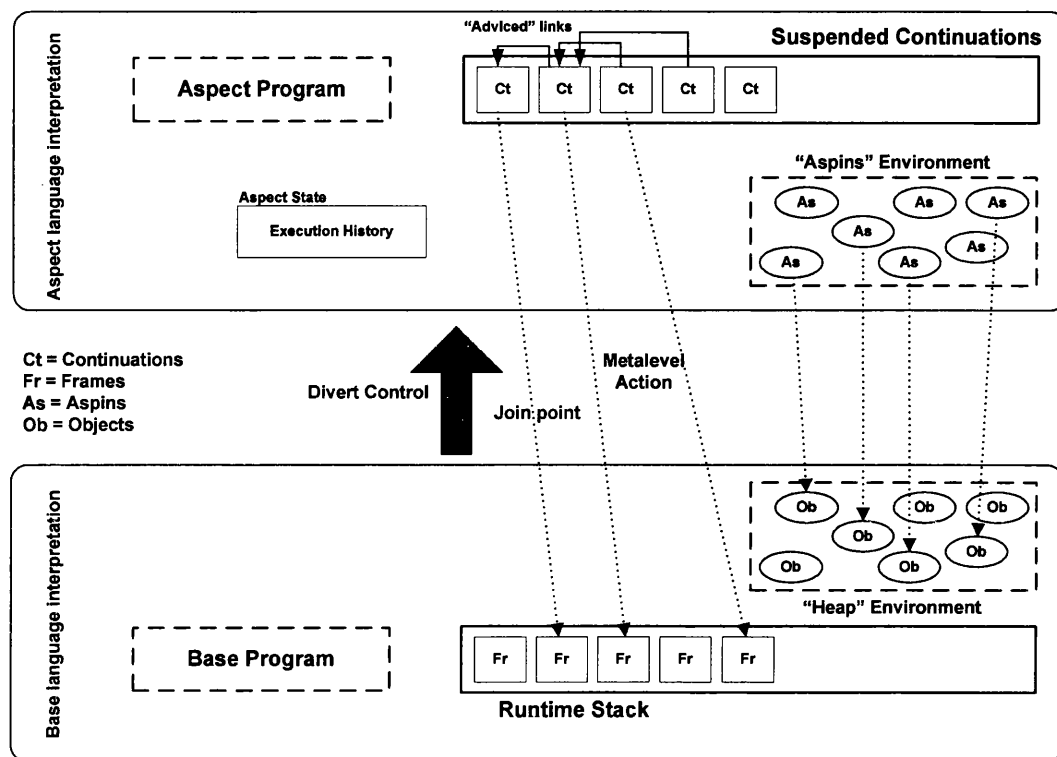


Figure 10 The base- and aspect-level interpreters of the metamodel from [54]

As shown in Figure 10 each time, only one executing program is the currently active continuation and all other continuations are suspended and saved. Each continuation (Ct) on this stack is a container for a set of frames (Fr) in the base interpreter and the continuation on top of the suspended continuations stack is actually the currently active continuation. The metalevel aspect interpreter creates a new continuation that represents the execution of the aspect's advice. It then schedules the execution of this continuation in the base interpreter that needs to execute the advice. When the base interpreter is restarted, it will thus first execute the advice. When the advice execution has finished, the aspect interpreter will re-activate the previous continuation on the stack. Each continuation also keeps a link to the continuation from which it was activated. This facilitates later manipulations such as the re-activation of the continuation at the join point from which the aspect was invoked. Each continuation is also activated again after the continuation that was switched to "has" finished executing. However, when a

program is halted at a join point, it effectively skips the evaluation of the expression that was scheduled to be executed at that join point.

Regarding metalevel operations, a metalevel action is an explicit join point, where control is given to the metalevel aspect interpreter. The ‘aspin environment’ in the metalevel interpreter allows keeping track of particular values for each different variable declared by all aspects. Upon execution of an aspect’s advice, the aspect program executes the StateSelector to retrieve the correct scope and the associated variable values [54, pp. 15-21].

3.6 Classification of Aspect Languages According to the Metamodel

The metamodel described earlier in this chapter, adapted from the [54], provides the foundation and the common understanding of the essential features of an aspect language. The metamodel was conceived in order to represent the commonalities and variations between aspect languages. Although the metamodel is a low-level aspect language in which other aspect languages can be expressed, often the metamodel need to implement specialisations in order to describe specific language features. Most of the aspect languages that the survey conducted in [50] had an object-oriented language as the base language. The aspect language that was chosen for the classification does not require implementing any specializations as such, because the language that it extends is not object oriented.

The survey did not cover AspectC [66] , a simple extension that adds AOP programming capabilities to C, because it was outside of the particular dimensions of interest of the partners of the network [50]. AspectC++ [67] was briefly covered in the survey but it was thought that it would be a useful exercise for this thesis to model AspectC for the following reasons. Firstly, in order to better understand the AOP capabilities and constraints that a developer may come across when trying to facilitate an AOP implementation in a procedural language [8]. Secondly the modelling will aid

the understanding of a case study [68] which is analysed in the next chapter. When that case study was first published, no framework had been defined that would allow an aspect language to be modelled in the way that the common language concepts metamodel would classify. Therefore, attempting to model AspectC can be useful for future researchers because it is another example that augments the usefulness of the metamodel even if in AspectC, aspects structure and modularize concerns that crosscut functions, files and directories rather than objects and modules.

For this purpose the metalevel aspect interpreter (metaspin interpreter) was developed to implement the metamodel which provides developers and researchers with a versatile aspect languages sandbox to be used for experimental classification of aspect languages and possible language integrations [54, p. 26].

Using AspectC as an example, this section will describe how the building blocks provided by the metamodel express the elementary features of an aspect language features. In other words how different aspect languages relate to the metamodel. Furthermore, it discusses the implementation of aspect languages in Metaspin based on the dimensions of an aspect language described in section 3.3. Note, that due the limitations of AspectC only join point, pointcut, and advice will be classified. More details about the rest of the dimensions can be found in [69].

Join Point Metamodel

The [50] identified the following categories of join point models [69, p. 6] :

Dynamic join points: All dynamic join point models fit the metamodel because the metamodel itself is completely based on dynamic join points.

Event-based (stateful) dynamic join points: These are identified as a sequence of events in the execution of the program.

Static join points: The metamodel itself is completely based on an interpreted semantics.

Domain-specific join points: The metamodel does not limit itself to a specific kind of paradigm but there hasn't been enough experimentation apart from with OOP. [68]

Classification into metamodel using Metaspin: Each join point is represented by a separate subclass of the join point class and then configured by a structural part (defined by the developer) and behavioural part (automatically represented by means of a continuation). In general, metaspin directly executes the methods for the mining of the behavioural properties on the join point class.

Figure 11 show how the AspectC join point model is classified in the metamodel through the use of the Metaspin Interpreter. AspectC intended to support operating systems and embedded systems programming [66]. It supports static join points (i.e. named entities in the program structure) as well as dynamic join points (i.e. events that happen during the program execution) [50, p. 24]. AspectC supports two types of join points: *function call and function execution* [66]. Pointcut functions are used to filter or select join points with specific properties. Some of them are evaluated at compile time and other at runtime [50, p. 24].

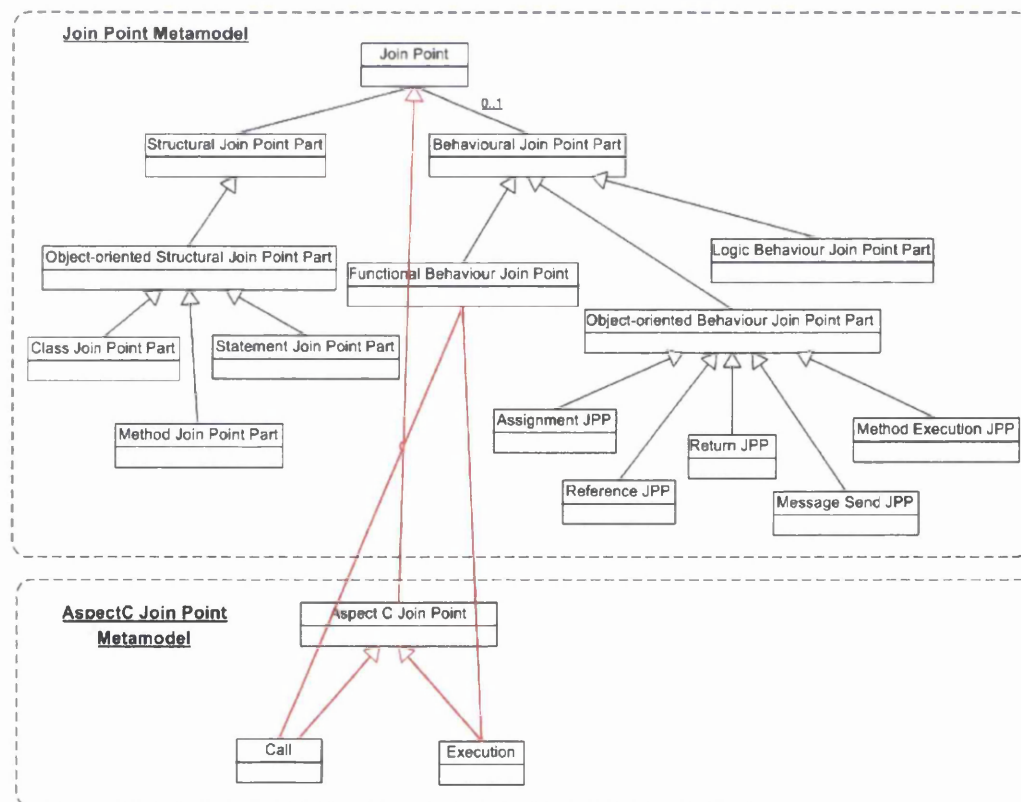


Figure 11 AspectC in the Join point Metamodel

Table 1 shows the pointcuts that AspectC can use for matching join points in terms of function signature and the type that would correspond to the metamodel [70, p. 26].

Table 1 Matching join points for AspectC

Syntax	Type	Comments
execution(Signature)	Current state predicate	Function execution join points signature matches Signature
call(Signature)	Current state predicate	Function execution join points signature matches Signature
base(Pointcut)	Query	Classes based on queries in the class hierarchy
derived(Pointcut)	Query	Classes based on queries in the class hierarchy
cflow(Pointcut)	Execution History	Captures all join points in the control flow of the join points specified by Pointcut
Within(File or Directory)	Current state predicate	Join points when the code executing is defined in one of the files found in File or Directory
that(Type pattern)	Current state predicate	Filters join points depending on the current object type
target(Type pattern)	Current state predicate	Filters join points depending on the target object type in a call
result(Type pattern)	Current state predicate	Filters join points depending on the result type of a join point
arg(Type pattern)	Current state predicate	Filters join points depending on the arguments type of a join point
Operators (!, &&,)	Composition Operator	Intersection, union, and exclusion of join points in pointcuts

Pointcut Language Metamodel

The main characteristics to classify a pointcut language are the following [69, p. 7]:

Language paradigm: The pointcut language paradigm is defined by the pointcut language evaluator.

Structural Properties: The definition of pointcuts is able to rely on structural properties of the source code. For that reason, a number of structure-reifying predicates can be offered in a pointcut language.

Behavioural Properties: The definition of pointcuts is able to rely on behavioural properties of the execution. For that reason, a number of predicates that reify dynamic and behavioural properties of the program are offered in a pointcut language.

Classification into metamodel using Metaspin: Because each pointcut language is specific to an aspect language the metamodel provides a common set of concepts for the classification of predicates and operators and also provides the interface through the

eval(JoinPoint) method. The implementation of the pointcut language must occur independently for each aspect language.

Figure 12 shows how the AspectC pointcut model is classified in the metamodel through the use of the Metaspin Interpreter. As mentioned already pointcut expressions determine the join points that need to be captured by the aspect. For the case of the metamodel, pointcuts correspond to their exact definition. For example as illustrated in Figure 12 the pointcut language paradigm is determined by the pointcut language evaluator. The definition of pointcuts can rely on structural properties of the source code behavioural properties of the execution [69, p. 8].

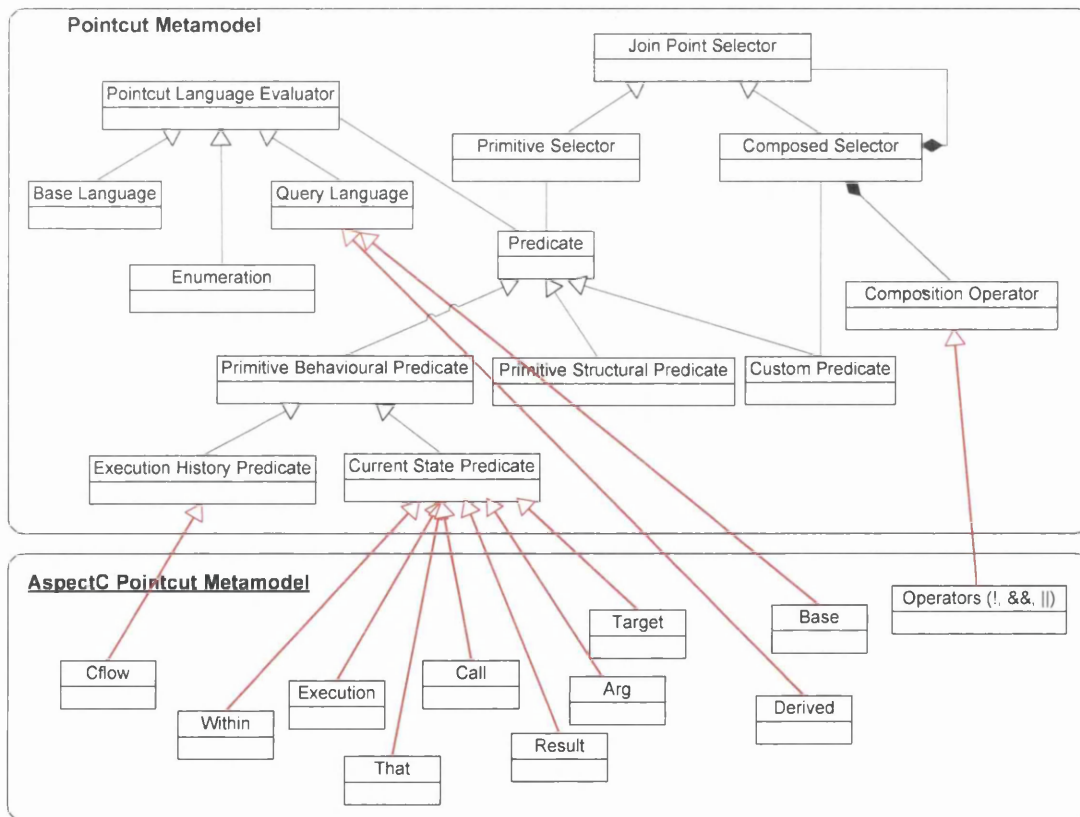


Figure 12 AspectC in the Pointcut Metamodel

Advice Metamodel

An advice in the metamodel [69, p. 9] has a number of actions that are executed instead of the join point by which it was triggered. Advices in the metamodel consist of expressions for the base (executed by the base language interpreter) and the aspect evaluator (executed by the aspect evaluator) and they can be mutually nested. The advice in the metamodel are *before*, *after*, *around*, *join point reflection*, *base language versus aspect-specific language* and *classification into the metamodel using metaspin*. In the last one (*classification into the metamodel using metaspin*) where an advice can implement a metalevel action as a message send to the metaspin class of which the selector is the same name as the metalevel action.

In terms of AspectC the only type of advice that is currently supported for static join points is the *introduction*. Using this advice the aspect code is able to add new elements to classes, structures, or unions. Dynamic join points use advice to affect the flow of control when the join point is reached. The types of advice that are supported are *before*, *after* and *around*. These advice types can orthogonally be combined with all dynamic join point types [50, p. 26]. Both after and around advice introduce additional special keywords such as the variable, *returned* for the after advice or *proceed* for the around advice. The *returned* variable accesses the return value of a function and the *proceed* variable explicitly requests execution of whatever would have run if the around advice had not been defined [70, p. 27].

Discussion

It can be seen that from this initial mapping of different language features into the metamodel that some improvements are required such as the syntax and structure of a language have not been taken into account in the metamodel. Although the initial metamodel was not intend to do that, structure and syntax have a significant impact on the expressiveness and identification of a language.

Furthermore, in the AOSD the Aspect Sandbox (ASB) [71] has similar approach with this work apart that the way that the interpreter execution semantics is considered without any weaving. ASB is a scheme interpreter to experiment with aspect-oriented

language features. The ASB provides a framework for building simple interpreters for AOP languages, together with implementations for a number of existing languages. Each interpreter models the semantics and implementation of one kind of AOP language. The framework is designed so that it is easy to understand the semantics of one AOP language in terms of what it adds to the underlying OOP language; to compare two AOP languages to each other; and to model the runtime costs of an AOP language construct [72]. The ASB focuses on the weaving semantics through the computation of join point shadows. On the contrary, the explicit setup of the metamodel and its interpreter is a complete interpreted execution.

3.6 Summary

This chapter attempts to set the foundations for reflection and analysis of AOP techniques. A survey of the current AOP technologies and frameworks was followed by an investigation of existing work on language models and meta-models for aspect-oriented programming which would allow the comparison and analysis of the fundamental aspect language features as well as their implementation and execution techniques. This was based on the results of an extensive survey was already conducted by the Aspect-Oriented Software Development (AOSD) community but the analysis was done with the pre-defined dimensions of interest by the partners rather than providing a complete overview of all language and execution model details. Nevertheless, because AOSD is an emerging paradigm that is trying to harmonise and integrate the research, train and disseminate the activities of its members in order to address fragmentation of AOSD activities in Europe and strengthen innovation in areas such as aspect-oriented analysis and design, formal methods, languages and applications of AOSD techniques in ambient computing [73]; AOSD Languages Lab goals were used to identify suitable AOP languages in terms of design space, implementation and runtime support technology.

Following the survey, an initial metamodel was conceived in the AOSD language labs and described from the survey results, the lessons learned from the survey and the extracted taxonomy of language features. Next, the results of the final dimensions of interest that reflect the essential concepts of an aspect language according to the AOSD Languages Lab were presented. These dimensions are the join point model, pointcut language, advice model and language, aspect module and composition model and aspect instantiation model. It was shown how the metamodel for aspect language is designed as an open-ended metamodel where the common concepts of aspect languages are represented and was explained that the open-ended property is of importance because it makes it possible to represent specific aspect language features through a translation of the specific aspect language features to the concepts in the metamodel and through a specialization of the common concepts in the metamodel.

Moreover, it was shown that the metamodel consists of a common model and an interpreter for instantiations of the common model. While the conceptual model describes the aspect language features, the metamodel interpreter implements their execution semantics. The common language concepts framework metamodel (common metamodel) were defined and explained in detail as four sub-metamodels namely the join point, pointcut, aspect binding and advice. The framework approach was taken in order to avoid oversimplification as specific language features of particular aspect languages can only be partially described as specializations of the concepts described in the common metamodel. An interpreter was also defined because the description of the semantics of the metamodel can be done by using the implementation of an interpreter. This is because the set of evaluation functions defined by the interpreter can have a close relation with its description using operational semantics. The interpreter can provide executable semantics which establishes a solid ground for tools to investigate and experiment with the semantics of language features.

Finally, it was described how different aspect languages can be expressed in terms of the metamodel. The initial experimentation was done using the metaspin interpreter, which is gradually reaching completion for further use in the languages lab. Using AspectC as an example, it was described how the specific language features that were identified in

the survey and the taxonomy can be modelled in terms of the metamodel. The resulting description allows modelling and classifying different aspect languages in the metamodel.

4. Assessing AOP – Approach and Implementation

The aspect-oriented approaches were developed based on certain instances of crosscutting code. Some examples of such approaches, implementations and models are: AspectJ, an aspect-oriented extension for the Java programming language that has been designed to be implemented in many ways [39] , [74]; a language framework for distributed computing [75]; synchronization policies [76]; database integration modelling using a composition-filters approach [77]; the specification of subject-oriented compositions [78]; and features such as multi-dimensional separation of concerns [79].

This chapter explains the criteria that must be met in order to assess AOP as a software technique that enables these approaches in practice. It begins with a discussion of the results of the research of two papers that answer the following questions:

1. How can one evaluate a new software development technique in terms of its usability and usefulness?
2. What are the typical factors that are required when evaluating a method?
3. What are the strengths and weaknesses of these evaluation methods?
4. How do developers manage when they encounter crosscutting code during a program change task?
5. What strategies are in place to deal with crosscutting concerns?

Further to the analysis, three different case studies were selected to analyse real world none trivial applications discussing the benefits and drawbacks of the AOP technique. The first case study provides a comparative analysis of the changes required to evolve the tangled and scattered versus aspect-oriented implementations. The second case study presents an AOP implementation of a classical example of crosscutting concern known as persistence. The third case study outlines how to conduct AOSD with use-case driven approach. The suggested solution is a new way of visualizing and capturing application and infrastructure use case flows while keeping infrastructure separate from the application and infrastructure services separate from each other.

4.1 Evaluation of Software Techniques and Management of Concerns During Evolution Tasks

4.1.1 Evaluating a Software Development Technique

The following section presents an evaluation of AOP from [81]. This explorative evaluation although limited, presents the lessons learned from two kinds of empirical study approaches (i.e. the use of a case study and experimental methods and the costs associated from them) with particular focus in assessing AOP. Some of the sources cited were found in the original research but investigated further. The presentation introduces the empirical study approach, summarizes the tools used, brief explanation of the case study, experiments and lessons learned. Further analysis and discussion was done, drawn from the results.

There are various ways that a technique can be evaluated. Murphy et al. [81] suggested making the technique accessible to the greater community and to see whether the approach sinks or swims but unfortunately this approach has drawbacks: useful techniques that are not yet usable can be lost, and usable techniques that are not particularly useful can inhibit the adoption of other, more powerful techniques.

Another approach [81, p. 2] is a form of empirical study that could include surveys, case studies, and experiments [82] and [83]. Empirical social research is commonly evaluated according to four tests [84]. These are construct validity, internal validity, external validity, and reliability. Construct validity refers to whether appropriate means of measurement for the concept being studied have been chosen; internal validity refers to how a causal relationship is established to argue about a theory from the data; external validity refers to the degree of generalization of the study; and reliability refers to the degree to which someone analyzing the data would conclude the same results.

However, direct application of these methods to studying software engineering questions is difficult. Therefore, many researchers are adopting variations of empirical techniques to assess development aids. These results can be found in [85], [86] and [87].

To evaluate the aspect-oriented approach, the Murphy et al. [81] decided to apply a three-month case study and a series of four experiments. This is because of the need to understand and characterize the kinds of information that each approach might provide when studying a technique that is in its infancy. The method for the case study [81] is based on the exploratory case study method described by Yin [84] reflecting on which aspects of the case study format proved useful, and which aspects of the format did not substantially help generate meaningful results. This was further complemented by domain-specific techniques. An example of domain-specific technique is to have lists of observational techniques that have been found to be useful for understanding the effects of the new software development approach on the development process [81, p. 3]. The experimental method is based on the human-computer interaction literature which has the same root as the experimental software engineering literature. The experimental methods were based on the human-computer interaction literature such as [88]. This literature has the same roots as the experimental software engineering literature such as [89].

Tools Used

Regarding the tools that were used in Murphy et al. [81], some of the design decisions are difficult to express cleanly in code using existing programming techniques. AOP is a new programming technique that intends to enable a more modular expression of these design decisions, which are known as aspects in the actual code [36, pp. 220-242]. AspectJ is used for the case study and experiments within the Microsoft Visual J++ environment running on Microsoft NT workstations. AspectJ uses a slightly modified form of Java, known as JCore and supports two aspect languages: COOL for expressing synchronization concerns and RIDL for expressing remote data transfer and method invocation concerns [90].

Case Studies

The case study method was used to answer two broad questions [81, p. 5]

- I. What types of programs are easier to write and change when using AOP?
- II. What effect does AOP have on software design?

The first question is regarding the usefulness of the technique and the latter on usability. Both questions are occurring with multi-person development environment. The case study set-up consisted of two phases:

Phase 1: Four interns developed a distributed game using AspectJ

Phase 2: Two interns re-developed the same application but using the traditional OOP approach and two interns implementing a distributed library application using AspectJ.

In summary, the results showed that AOP approach was particularly useful when the aspect language matched a design concern, such as concurrency because the language provided a vocabulary for expressing and reasoning about that concern but an increase in design complexity, when a particular aspect language is used to try to express a concern not intended by that aspect language [81, p. 6]. Furthermore, it helped realise potential challenges of the usefulness of AOP in other settings; improve the usability of the approach by providing a concrete set of language features; a number of potential research directions [91].

Experiments

After the use of the case study method to evaluate usefulness and usability, four experiments were set to examine three specific tasks in order to understand how AOP can act as a catalyst for particular programming tasks [81, p. 14].

The experiments are:

- I. Comparison of OOP versus AOP in terms of the ease of creating a program.
- II. Comparison of OOP versus AOP in terms of ease of debugging.
- III. Comparison through investigation in terms of ease of changing an OOP versus AOP program.
- IV. Investigation of a combination of these activities.

The experiments were considered as semi-controlled empirical studies due to constraints by small number of participants, time shortage, high costs in relation with running and analyzing experiments and forfeit precision of measurement in favour of realism [92].

Table 2 depicts an overview of all the experiments including details about its set-up and results in terms of development man hours.

The experiments were successful in gathering qualitative but sometimes supported with limited quantitative evidence about the usefulness of AOP helping into revealing which parts of the approach contribute to its usefulness and usability. More detail about the experimental setup and results can be found in [93].

Experiment	Description	Experiment Set-up	Results (Hours)	
			OOP	AOP
Pilot Study	Can a developer produce an AOP working multi-threaded program in less time, and with fewer bugs than OOP?	Small programming problem with concurrency	3	4
Debugging	Can the ability of a user to find and fix functionality errors (bugs) present in a multi-threaded program enhanced by the separation of concerns in AOP?	Three cascading synchronization into an approximately 600 line digital library program	3	3
Change	Comparison through investigation in terms of ease of changing a program.	Add same functionality into a 1500 line distributed digital library Tools: OOP: Emerald distributed O-O language AOP: RIDL, COOL	4	4
Combination of activities	Developers working independent applications using AspectJ	Substantive changes to a skeleton of a program	n/a	8

Table 2 Experimental Methods Overview and Results

Lessons Learned

The paper presents some of the highlights of the overall assessment lessons learned so far [81, p. 22] which are divided into three areas:

1. Selection of an evaluation method.
2. Areas to which particular attention must be paid to maintain realism.
3. Issues that may arise in designing either a case study-based or experimentally-based empirical evaluation.

1. Selection of an Evaluation Method

The choice of the method is based on the degree of control an investigator has over the environment in which the study is conducted. The spectrum of the choice method starts with case studies which exerted less control than combinative experimental method which exerted less control than our comparative experimental methods. [81, p. 23] What questions must be considered if this method was to evaluate a new software engineering technology. For example, what elements of the technology does the researcher need to know? What is the budget (time and cost) for the evaluation? What are the expected results? [81, p. 23]

In terms of goals of the evaluation, a case study approach was more effective if the primary interest is in the broad effects of the new technology. This approach gives the ability to gather data from diverse areas such as design processes or environment problems. The combinative experiment was also used to gather similar qualitative data about multiple facets of tasks in a more controlled setting but it wasn't as broad. The case study approach was more effective because it quickly identified and addressed the usability issues with the technology. Furthermore, it allowed sufficient flexibility for the developers to have a range of interaction with the technology. An important question when evaluating a technology is to decide whether it is reasonable to try to address concurrently usefulness and usability. Because usefulness and usability are closely tangled for new technologies, determining how to investigate them together or how to separate these issues at reasonable cost is important. [81, pp. 23-24]

Selecting a method also requires consideration of the stability of the technology. The greater the control that is desired in a study, usually the greater the investment that is required in preparation time and labour costs. For the sake of stability it is helpful to maintain the programming environment versions consistent over a course of evaluation for result comparison. Furthermore, new versions could introduce more problems or bugs to the current implementation of the case studies. As mentioned evaluation cost is also an issue, particularly for technologies that are rapidly evolving. Finally, regardless of the chosen method the appropriate balance of construct validity, internal validity, external validity, and reliability is necessary. The paper suggests that none of the

methods used achieved the desired balance easier than any other. [81, pp. 23-24]. Therefore, choosing a method should depend on the feasibility of conducting a study given the budget is available for the questions of interest.

2. Maintaining Realism

Maintaining a reasonable degree of realism is a difficult task [81, pp. 25-26] while investigating how a new technology can help the process of software development. For example, how can a case study balance strict time constraints while is trying to tackle a serious problem arising in software development? The “time” issue is even greater in the context of experiments when selecting appropriate problems (motivating to the participants and reasonably realistic) that developers could tackle. It is difficult to provide general guidelines on how to approach the problem selection problem for experiments apart from suggesting dress rehearsal (trials) and planning in order to ensure that the problem is manageable. Realism can be introduced into the environment by letting developers interact as much as possible with the tools, settings and the development environment. Finally, the skill set and experience of the developers is an important factor for results expected to achieve.

3. Designing the Empirical Study

Further to the guidance already suggested earlier on with particular emphasis on experimental studies for software engineering from [89] and [82] are data gathering and analysis. Gathering meaningful data about a task i.e. trying to achieve the construct of validity is a difficult task. Performing these kinds of tasks involves problem solving at abstract and concrete levels [94] , time management, and communicating ideas, among other activities. Finally, determining what data analysis is required before conducting experiments and case studies is ideal but difficult to put it in practice because the data analysis strategy is usually not clear at the start of a project.

Conclusion

Validity, realism and cost are typical factors that are required when evaluating a method that helps software development. The flexibility in each of these factors increases with the maturity of the technology. Two methods were used to study AOP, namely case study and experiments. Because the paper was written at the time that AOP was relatively a new technology these methods were more exploratory.

The case study method provided results about the usefulness and challenges of the technique, concrete features that could improve the usability of the approach, and about potential research directions. The experimental approach provided qualitative evidence about the usefulness of the technique and identified more specific parts of the approach that contribute to its usefulness and usability. Overall, the case study was more effective means of achieving our initial goals of assessing whether and how AOP might ease some development tasks. Regardless of the results it is important to note that AOP is not trying to replace OOP but to capture important design decisions that are difficult to capture in the traditional OOP (i.e. a new programming technique) [36]. Therefore, the experiments although exploratory, would yield better results if better focussed on issues such as crosscutting concerns.

The paper [81] makes two contributions. First, analyzes the costs of applying several different evaluation methods highlighting some strengths and weaknesses of the various approaches and introducing data gathering and analysis method particularly on experimental studies. Second, discuss the possible value of various forms of semi-controlled studies particularly in new technologies. These studies can help determine if the technique shows promise, and whether it can help direct the evolution of a technology to increase its usability and potential for usefulness.

4.1.2 Managing Crosscutting Concerns During Software Evolution Tasks

The code of an application is modularized as a mechanism for improving the flexibility, efficiency, extendibility, reusability and comprehensibility of a system while allowing the shortening of its development time [35]. AOP provides support on design decisions that the program must implement but are hard to express them clearly with a modular fashion because they crosscut the systems basic functionality [36]. The aspect-oriented

approaches were developed based on certain instances of crosscutting code. Examples of such approaches, implementations and models at the time that the paper was conducted are: HyperJ, a multi-dimensional separation of concerns supporting construction, evolution and integration of software [47]; AspectJ, an aspect-oriented extension for the Java programming language that has been designed to be implemented in many ways [39] , [74]; Language framework for distributed computing [75]; Synchronization policies [76]; Database integration modelling using a composition-filters approach [77]; Specifying subject-oriented compositions [78]; Features such as multi-dimensional separation of concerns [79].

There have been few papers introducing evaluation and empirical methods that provided results on the usefulness and challenges of AOP [81] or discussing the effect of aspects on object-oriented development practices [95]. But there haven't been empirical studies to consider the various crosscutting concerns that developers would find beneficial to modularize, or how are developers currently managing those concerns in existing systems. The presentation of the study [96] aims to gain an insight on these concerns by studying the progression of eight developers from industry and academia on a change task. Each developer was making non-trivial changes to different non-trivial applications. The data analysis results showed that each developer had to consider at least one crosscutting concern that arose when encountering problems in making their desired change. For example, a developer encountered security issues, communication protocols and hardware platform dependencies concerns when trying to change the mathematical model applied to a specific new purpose. In order to manage these issues three solutions emerged depending on how the concern interacted with the core code associated with the change: (1) change the entire concern, (2) work within the conventions of the concern, (3) alter the change task rather than coping with the concern.

Furthermore, the results of this study [96] provides with:

- Empirical evidence about the kinds of crosscutting concerns that impact software developers

- The strategies developers use to cope with these kinds of concerns in existing systems.
- A comparison basis in order to answer whether the use of aspect-oriented approaches enables developers to better represent and work with crosscutting code. In other words, does the use of AOP eliminate the need to alter a change task in situations similar to those described by this paper?

As with the previous case study, the approach that was taken to present this case study [96] is to briefly explain the setup of the experiment and its outcome followed by a discussion on the implications of the results. Also, few of the sources cited were found in the original research but investigated further.

Setup and Tools Used

The duration of study method was three weeks and used interviews as its main tool, based on the data collection methods for software field studies [97]. Eight separate change tasks were considered, each performed on a unique system. The systems were implemented in range of programming languages: three systems were implemented in C [98], three in C++ [99], and two in Java [100]. The tasks were implemented by eight participants, four senior developers of which two had prior AOP experience, and four graduates with generic programming experience. An important requirement of the study was that participants would have limited prior knowledge of the code base and therefore would have to investigate the scope of the change. This was achieved by having them working on an application that they weren't the initial or a principal developer.

The information that was required to be gathered through the series of these three one hour interviews was: the program change of the developer, the approach to the task, the approach to determine which segments of code needed to change, and the degree of difficulty to make the change, if so, why it was difficult.

As mentioned, the main focus of the study [96, p. 2] was determining the kinds of crosscutting concerns that developers must consider in existing code bases. The approach that was taken was by asking questions about the change task rather than directly about the concerns. This approach was taken because it showed through the

interviews that most of the developers hadn't thought about crosscutting concerns. They didn't understand the meaning of the questions when asked directly about these concerns not to mention that some with prior AOP knowledge would just answer with popular crosscutting concerns like tracing, debugging, or distribution and therefore, could have hidden other concerns related to the task. Finally, it took time for the developers to think about the problem in broader terms because of their heavy involvement in the details of the task. As the interviews progressed the developers started to think about their tasks in a more conceptual level which allowed them to consider more high level questions. This led to aid them to indentify portions of the code that they would like to see modularized.

Results

Most developers described their change task from two perspectives: a structural perspective and an obstacle based perspective [96, pp. 2-5].

Looking into the straightforward structural perspective, it can be seen from the initial description of the developers that their change task was easily identifiable structure in the code. They described the change in terms of a particular data structure or a particular module in the code which was straightforward but often scattered. They could understand the purpose of the code and its context within the structure of the application and point out portions of the code that corresponded to their change, but only the developer with prior AOP knowledge described crosscutting code as the target of the change.

In terms of the non-straightforward obstacle perspective, the developers realised that although they knew the locations in the code that needed to be changed, they faced a set of obstacles when making the change. The obstacles comprised segments of code that were relevant to the task but that also affected an underlying concern; this code was at the intersection of the core change and the broader concern. Hence, in order to make the change the developer had to understand the entire concern and since that underlying concern was not well-modularized or well-documented, it was difficult to conceptualize and to reason about [96, pp. 2-5].

Table 3 summarizes the program change tasks, the obstacles faced and the strategy employed for each developer.

Three strategies were used to deal with the obstacles:

- I. Change: Alter the concern code to enable the change task.
- II. Within: Understand the concern associated with the obstacle but not changing sufficiently to make the change work within the concern.
- III. Around: Completely alter the change task to account for the concern without understanding the concern.

Examining how participants addressed the obstacles they faced and focusing on the obstacle points the locations at which the change task intersected the crosscutting concern, it was found that there were certain patterns of interaction between the concern and the change code. It was determined that there was a relationship between the patterns and the strategy to address the obstacle. [96, pp. 2-5]

Table 3 Developers task descriptions, obstacles and strategies

Developer	Straightforward Structural view	Non-straightforward obstacle view	Strategy
1	Moving particular computation to an aspect-like module	Synchronization Performance	Within
2	Tailoring a matching algorithm for a specific purpose	Memory allocation	Change
3	Changing matrix calculation	Memory allocation	Around
4	Changing Table representation	Implicit assumptions about data structure representations	Around
5	Changing packaging of user interface mechanism	Distribution, Tracing	Within
6	Changing the mathematical model applied	Security issues Communication protocols, Hardware platform dependencies	Within
7	Changing printing look and feel	User Interface consistency, Printing speed	Change
8	Adding cancellation notification to an existing system	Multithreading, Behavioural consistency	Within

Change strategy - changing the relevant portions of the crosscutting concern to suit the change: The change strategy had a structural intersection point. The developers could identify, from the code related to the change, certain structures such as types, objects, and computations directly related to those structures as obstacles to their change task. I.e. these obstacle points provided enough information about the broader concern to lead the developer reason the points of change, located in the broader concern. Developer seven was more visible because the changes were at the user interface level. Developer two was able to estimate that all functionality of a certain kind involving a particular type would have to be altered. It was then straightforward, though tedious, to make the changes.

Within Strategy - understand the effect of the code on the crosscutting concern that presents an obstacle to the change, and work within the conventions of the concern: The within strategy, followed a behavioural pattern. The intersection of the change code and the behavioural concern code could not be assessed as easily as the structural case, because the obstacle points were implied. The developers had to examine the broader concern in order to understand the conventions of the concern and then had to reason inward about how to change the core code to work within the broader concern. Essentially, they had to gain a general understanding of the code base in order to work within the concerns. Once they had this understanding, they were able to identify portions of code that would allow them to reason inward about their specific change task. Developers one, five, six and eight used this strategy. It is worth mentioning that developer eight had to perform considerable testing to ensure the obstacle had been dealt with appropriately.

A good example of inward reasoning is the attempt of developer one to move pre-fetching functionality within operating system code into a separate aspect like module. The developer knew that this change would impact synchronization in the system and had to reason inward from the synchronization concern to the core change. A suggested solution was to include synchronization code in the new pre-fetching module even though the code was not directly related to the core of the change. The inclusion of this

code ensured that the locking invariants encoded in the synchronization concern were maintained. The study of aspect evolution in operating system code by [68] discusses further suggestions on how concerns such as pre-fetching can better modularized using aspect-oriented implementation. In all cases, however, developers were unable to cleanly determine when they had addressed all of the code related to their change.

Around Strategy – a significant rethink of the original approach to change task because of the developers lack of understanding for obstacles, and not being able to address the concern: The around strategy, was dense. The code made ambiguous use of assumptions from around the code base and was thus subtle and difficult to reason about. When the change approach became too difficult, the developers were forced to work around both the obstacle and the concern code. The obstacles associated with the strategy are encoded, meaning that they are neither structurally explicit, nor are they implied by comments or conventions. As a result, the developer was unable to use either of the inward or outward reasoning strategies employed by other participants. In the end, the participant simply worked around this difficult code. Developers three and four used this strategy i.e. each worked around the obstacle.

It worth mentioning how developer four ran into memory allocation problems after making what should have been a simple change. After failed attempts to understand how the change affected the memory allocation for the application, a work around was devised to trick the memory allocation portions of the source into thinking that the change had not been made.

Result Implications

The results showed that a significant effort was required by the developers to understand the segments of the crosscutting concern associated with the obstacle. It was not an easy task to determine the connection between the code related to the change and the broader concern especially for those who used the within strategy i.e. when the developers were considering the code related to the change they had to ask themselves how the crosscutting concern will be affected if this location in the code is changed.

The outcome of the study [96, pp. 5-6] is an empirical evidence of crosscutting concerns and the strategies used in coping with such concerns, because of the similarities in the form of the crosscutting code involved, and in the strategies used by the participants to cope with these concerns despite the differences in developers, tasks and systems. These similarities are indicative of real software developments and allow results to generalize. The outcome also showed that AOP can help avoiding the around strategy by modularizing a particular crosscutting concern.

However, because of its exploratory nature, the study was limited to a small number of systems, tasks and time constraints. An important observation is that prior to the study it was assumed that concerns might be more directly linked with change tasks, i.e. a change might correspond with a concern, but the study showed that concerns typically intersected changes. Further testing is needed to see if it is imperative that concerns intersect change.

This study can compare with other empirical work in two areas: (1) studying the way developers perform software change task and (2) examining AOP.

Developers study: A lot of work has been done to analyze the cognitive and thought process approach that developers use to understand code. These approaches can be characterized as

- I. Top-down [101], [102], where the developer begins with understanding of a general nature.
- II. Bottom-up [103], [104], where developer begin by reading source code and by mentally forming higher-level abstractions.
- III. Knowledge-based [105] which involves incorporating domain knowledge and the mental models formed during program analysis.
- IV. Integrated [106], which incorporates all of the above.

The above approaches are focussing on work practices and the models built by developers while understanding the code. The study is focussed on the form and role of the code that developers examine when performing a program change task.

Examining AOP: The study showed that when performing a task at certain points the developer needed to see the behavioural effects of aspects on methods of interest. Another point was regarding the controlled experiment to investigate whether AOP could ease program maintenance tasks. The results showed that developers found it difficult to reason about a separated concern when the interface between the core code and the concern code was too broad i.e. the more constrained and defined the interface, the easier it was for developer to determine the area of influence between the code and concern code. This result was also verified by [107].

4.2 Case Study I: A Retroactive Study of Aspect Evolution in Operating System Code

Overview

Operating Systems (OS) must perform well under an increasingly diverse set of workload demand. But evolving OS code is hard because it involves extending, integrating, optimizing, re-optimizing, and maintaining system functionality. It not only requires understanding the individual concerns within the system, but often their inherently complex interactions.

As mentioned modularity helps evolution by providing a shorten development time because separate groups would work on each module with little need for communication, the possibility of making drastic changes to one module without a need to change others and the ability to study a system one module at a time i.e. the entire system can therefore be better designed because it is better understood. But providing a clear division of responsibilities in OS code is hard and many studies such as [108] has shown that the average number of modules involved in a change rose significantly in new releases due to unintentional interaction among modules.

This case study [68] describes the impact evolution had on these concerns, and provides a comparative analysis of the changes required to evolve the tangled versus aspect-oriented implementations. Coady [68] suggests that AOP can be used to improve

evolvability of OS code by providing better modularity of crosscutting concerns and interacting concerns without harming non-interacting concerns. The results show that for the concerns that were explored, the aspect-oriented implementation facilitated evolution in four key ways:

1. Changes were better localized
2. Configurability was more explicit
3. Redundancy was reduced
4. Extensibility aligned with an aspect was more modular

The experiment [68] was set up on FreeBSD v2.2.8 Dec 1998, v3.3, Sep 1999 and v4.4 Sep 2001. FreeBSD's development began in 1993 and grew into an operating system taken from U.C. Berkeley's 4.4BSD-Lite. FreeBSD is representative of a high quality implementation of an operating system because of its design lineage in the research community and successful adoption in industry. It is one of the most widely-distributed Unix-based operating systems and because of its open source nature; FreeBSD is an excellent platform for research in operating systems as well as other branches of computer science. FreeBSD's freely available nature also makes it possible for remote groups to collaborate on ideas or shared development without having to worry about special licensing agreements or limitations on what may be discussed in open forums [109].

The FreeBSD operating system was doubled in size in terms lines of code (LOC) between version 2 and 4 (i.e. during the span that the research took place). Changes to primary modularity at high-level such as new device drivers are easier to trace than changes to crosscutting concerns such as the number of places where disk quotas are tracking disk utilization enforcing limits to users. Four crosscutting concerns were refactored in version 2 of the code into aspects [36] in order to better understand how an aspect-oriented implementation works from the view of system evolution [68]. The concerns are waking the page daemon, pre-fetching for mapped files, quotas for disk usage, and tracing blocked processes in device drivers. These implementations were then rolled forward into their subsequent incarnations in versions 3 and 4 of FreeBSD. This paper describes the impact evolution had on these concerns, and provides a

comparative analysis of the changes required to evolve the original versus aspect-oriented implementations through a range of scenarios. [68]

The approach that was taken to review this research [68] is as follows, each crosscutting concern is analysed individually starting from an understanding of its nature, overview of the original implementation, changes required to evolve the aspect-oriented implementation followed by a comparison with the impact of evolution on the original tangled implementation. After summarizing and analysing all the concerns, a collective analysis that reviews the results including a brief introduction of the costs associated with the AspectC runtime is discussed. As with the previous research papers, some of the cited sources were found in the original research but have been investigated further. Also, the sample source codes are taken from the research but further comments have been added, as it hoped to illustrate the AOP implementation approach in practice for non-trivial applications.

Analysis of the Crosscutting Concerns

1. Page Daemon Activation concern

A page or virtual page is a fixed-length block of main memory that is contiguous in both physical and virtual memory addressing [110, p. 32]. When the number of available pages falls below a certain threshold page daemon is designed to be activated in order to assess where is needed to free physical memory. Determining which pages will be replaced and writing them back to disk if necessary imposes overhead and therefore timing is important as the daemon should be activated only when required. The structure of page daemon activation is a set of context-specific triggers within the virtual memory system and the file buffer cache. Therefore, the activation crosscuts operations that consume available pages [68, p. 50]. In the original implementation the function `pagedaemon_wakeup()`, and its lower level counterpart, `wakeup (&vm_pages needed)` are invoked less as moving from version 2 to 4. Triggers for page daemon wakeup were eliminated as the virtually memory (VM) system evolved and many functions such as (`swap_pager.c`) were significantly revised. Finally from version 3 to 4 the function to

allocate pages was reworked, which resulted in eliminating further triggers. In addition the introduction of a new synchronization operation in VM added a new low-level activation of the daemon [68, p. 51].

Three main changes had to be done to the code to evolve the aspect-oriented implementation. The first was introducing page daemon activation as an aspect using AspectC. This involved re-factoring and removal of code that controlled activation from the operations it crosscut. The second change was regarding internal structure and implementation. The page daemon use thresholds to determine when activation is required. Therefore it imperative to understand for the entire system the contexts of the threshold checks, the specifics of the thresholds used, and the relationship between the contexts and the thresholds. The below code extract shows some of the core implementation of the page daemon wakeup aspect common to all versions. [68, p. 52]

```
aspect page_daemon_wakeup {
// pointcuts identify specific points in kernel execution when paging may be needed i.e. when
//unqueueing available pages & when allocating buffers
    pointcut unqueueing_available_pages(vm_page_t m):
        execution(void vm_page_unqueue(m))
        && cflow(execution(void vm_page_activate(vm_page_t))
        || execution(void vm_page_wire(vm_page_t))
        || execution(void vm_page_unmanage(vm_page_t))
        || execution(void vm_page_deactivate(vm_page_t, int)));
    pointcut allocating_buffers(vm_object_t obj, vm_pindex_t pindex):
        execution(vm_page_t vm_page_lookup(obj, pindex))
        && cflow(execution(int allocbuf(struct buf*, int)));
// advice declarations use pointcuts to associate a given page threshold with a point in the
// execution of the system and wake the daemon accordingly
    around(vm_page_t m):
        unqueueing_available_pages(m) {
            int queue = m->queue;
            proceed(m);
            if (((queue - m->pc) == PQ_CACHE)
                && (pages_available() < vm_page_threshold()))
                pagedaemon_wakeup();
        }
}
```

```

    }
    around(vm_object_t obj, vm_pindex_t pindex):
        allocating_buffers(obj, pindex) {
            vm_page_t m = proceed(obj, pindex);
            if ((m != NULL) && !(m->flags & PG_BUSY)
                && ((m->queue - m->pc) == PQ_CACHE)
                && (pages_available() < vfs_page_threshold()))
                pagedaemon_wakeup();
        }
    return m;

```

The aspect was impacted from evolutionary changes such as the swap pager, the VM page operations, and page daemon activation code. The changes involved further refactoring, adding or deleting specific pointcuts and advice, and introducing some helper functions to reduce redundancy and increase readability. [68, p. 52]

2. Pre-fetching concern

The inherent structure of pre-fetching is shaped by specific execution-paths that retrieve pages from disk, which is an expensive operation [111, IBM]. Pre-fetching is a heuristic meaning that it is designed to reduce the costs by securing additional pages that may be required but it is important to pre-fetch only when it is cost effective. Pre-fetching crosscuts virtual memory and file systems, coordinating high level allocation and low level de-allocation of pre-fetched pages [68, p. 51]. The study regarding managing crosscutting concerns during software evolution tasks by [96] discussed how a developer struggled when stumbled on a concern such as the pre-fetching functionality, and worked hard to understand the effect of their code on the crosscutting concern that presented an obstacle to their change. This resulted working within the conventions of the concern i.e. to reason inward from the synchronization concern to the core change. It is hoped that aspect-oriented implementation presented in this study aid the future implementations of this kind.

The original implementation of FreeBSD had some changes between version 2 and 4 such as files changes, levels of function Tables, variable names [112]. The most significant change was between version 2 and 3, where the sequential mode pre-fetching

in one file system was modified to be more aggressive. However this was removed in version 4.

In terms of the aspect-oriented implementation the code was re-factored like the page daemon activation concern in order to introduce the pre-fetching concern as an aspect for mapped files and allow for more fine-grain composition. The re-factoring did not require further modification as the versions evolved [68, p. 53]. The below code extract shows a portion of the implementation of the single aspect for normal and sequential mode mapped file pre-fetching common to versions 2 and 4 [112].

```
// This aspect structures the coordination between the high-level allocation and their possible  
//subsequent low-level de-allocation for pre-fetched pages
```

```
aspect mapped_file_prefetching{
```

```
// The pointcuts name the high-level (vm_fault_path) and the low-level (getpages_path) parts of  
the //execution paths involved
```

```
pointcut vm_fault_path(vm_map_t map):  
    cflow(execution(int vm_fault(map,...)));
```

```
pointcut getpages_path(vm_map_t map, vm_object_t obj,  
                      vm_page_t* plist, int n, int fpage):  
    cflow(execution(int ffs_getpages(obj, plist, n, fpage)  
|| execution(int vnode_leaf_pager_getpages(obj, plist,n,fpage)));
```

```
// The advice coordinates allocation (before) / de-allocation (after) pages for pre-fetching
```

```
before(vm_map_t map, vm_object_t obj, vm_page_t* plist, int n,  
       int fpage):  
    execution(int vnode_pager_getpages(obj, plist, n, fpage))  
    && vm_fault_path(map)  
    { ...plan and allocate prefetched pages...}
```

```
after(vm_object_t obj, vm_page_t* plist, int n, int fpage, int valid):  
    execution(valid check_valid(..))  
    && getpages_path(obj, plist, n, fpage)  
    { ...dealloc all prefetched pages...}
```

```
after(vm_object_t obj, vm_page_t* plist, int n, int fpage,  
      struct transfer_args* trans_args):  
    execution(int calc_range(trans_args))  
    && getpages_path(obj, plist, len, fpage)
```

```
{ ...dealloc non contiguous pages... }  
...}
```

The aspect was not impacted from evolutionary changes however changes were done on the concern itself. The aspect was split in two parts with the introduction of sequential mode between version 2 and 3 and then became one after the removal of the sequential mode in version 4 of the original implementation [68, p. 53].

3. Disk Quotas Concern

The disk quota system provides an effective way to control the use of disk space [113, Publib boulder IBM]. The inherent structure of quota is a set of low-level disk space related operations that consistently monitor or limit all disk usage. Quota crosscuts operations that consume and free disk space in file systems that offer support for this functionality. Because disk quotas are an optional feature of FreeBSD the original implementation was conFigured through a combination of settings in both a kernel configuration file and on a per-file system basis [68, p. 51].

In terms of the aspect-oriented implementation re-factoring quota in version 2 involved separating the segments of quota code associated with compiler directives from the file system operations it crosscut. This allowed for composition of the aspect with the precise granularity of file system functionality it crosscut. Second extended file system (EXT2), file system for the Linux kernel, introduce identical functionality to the corresponding operations in the union of Unix File System (UFS) and fast file system (FFS). That is, all the quota code in EXT2 is redundant. The below code extract shows a section of the implementation of the disk quota common to all versions [68, p. 54].

```
aspect disk_quota {
```

```
//The pointcuts name the corresponding operations from the different file systems that are  
//associated with shared quota operations.
```

```
    pointcut flushfiles(register struct mount *mp, int flags, struct proc *p):  
        execution(int ffs_flushfiles(mp, flags, p))  
        || execution(int ext2_flushfiles(mp, flags, p));
```

*//Around advice that uses this pointcut provides a single shared implementation of the
//associated quota operation*

```
around(register struct mount *mp, int flags, struct proc *p):
    flushfiles(mp, flags, p) {
        register struct ufsmount *ump;
        ump = VFSTOUFS(mp);
        if (mp->mnt_flag & MNT_QUOTA) {
            int i;
            int error = vflush(mp, NULLVP, SKIPSYSTEM|flags);
            if (error)
                return (error);
            for (i = 0; i < MAXQUOTAS; i++) {
                if (ump->um_quotas[i] == NULLVP)
                    continue;
                quotaoff(p, mp, i);
            }
        }
        return proceed(mp, flags, p);
    }
```

For versions 2 and 3 the aspect was impacted from evolutionary changes such as the introduction of a new feature for file servers, the implementation of compiler directives and between versions 3 and 4 a new FFS operation was introduced requiring quota tracking. This primarily consisted of adding pointcuts and advice as needed to incrementally extend its configuration to include new functionality [68, p. 54].

4. Blocking in Device Drivers Concern

Scheduling is a key concept in computer multitasking and multiprocessing operating system design, and in real-time operating system design. Scheduling is the way processes are assigned priorities in a priority queue. Scheduling is concerned with tasks such as keeping the CPU as busy as possible (CPU utilization), the number of process that complete their execution per time unit (Throughput), the amount of time to execute a particular process (Turnaround), The amount of time a process has been waiting in the ready queue (Waiting time), the amount of time it takes from when a request was submitted until the first response is produced (Response time) etc. Furthermore, different computer operating systems implement different scheduling schemes [114].

Blocking in device driver code is designed to maximize CPU utilization while processes wait for device I/O. The structure of diagnostic statements related to blocking behaviour in device drivers shadows all points in the system where a process could be blocked on a device indefinitely. Tracking process blocking in device drivers crosscuts all device-specific operations involved with I/O [68, p. 51]. When waiting for device I/O, a process blocks by calling `tsleep()`. Functions such as `tsleep()` or `ltsleep()` implement voluntary context switching and are used throughout the kernel whenever processing in the current context cannot continue for reasons like when the current process needs to await the results of a pending I/O operation or a process needs resources (e.g., memory) which are temporarily unavailable etc [115]. The `tsleep()` is passed a value to block on and a timeout after which the process will wake-up using the `wakeup()` if it has not been unblocked [68, p. 52].

In the original implementation the device driver code has the highest rate of growth and therefore the highest rate of bugs in the kernel [116]. An aspect was introduced in the version 2 of the driver code to track processes that block on device operations without a timeout. As processes may block indefinitely, diagnosing problematic behaviour associated with device drivers can be of particular interest. Although the number of calls to `tsleep()` in driver code grew from 5, to 55 and 110 in versions 2, 3 and 4 respectively, the only modification required to evolve this aspect was to make one the functions parameter constant between versions 3 and 4.

Analysis of the Results of the Experiment

The results are grouped into four areas:

1. The ways in which the original evolution of each concern was problematic is overviewed.
2. The general ways in which the aspects addressed these problems are summarized.
3. A brief overview of cost analysis associated with runtime support for the aspect-oriented implementation.

4. Open Issues

1. Evolving Scattered and Tangled Code

The concerns discussed earlier are non modular, scattered and tangled in an unclear way throughout the primary modularity they crosscut in their original implementations. The specific problems that developed during the evolution of the original implementation of each concern are summarized here.

Page Daemon Wakeup

Identifying exactly in the system when and why this concern should be activated is important but also difficult because the code is spread out. This is why some of the activations were re-factored, while others were not. It is also evident that is imperative to understand for the entire system the contexts of the threshold checks, the specifics of the thresholds used, and the relationship between the contexts and the thresholds. The subtle differences are critical for understanding daemon activation, but difficult to appreciate in the original implementation. For example page fault handling has the only threshold check that does not use `cache_rain` which is the minimum number of pages desired on the cache queue. Finally it seems that VM and the buffer code did not evolve simultaneously because re-factoring of the threshold calculations were included but not applied consistently to all the thresholds involved with activation [68, p. 55].

Pre-fetching

Although there were only small changes to the system because of the addition of the sequential mode in version 3 it introduced the relationship between VM, the file system, and the buffer cache that did not exist previously [68, p. 55].

Disk Quotas

Implementing quota with pre-processor directives supports efficient, coarse grained configurability. Pre-processor directives are not program statements but directives for

the pre-processor (preceded by hash sign #) and are executed before the actual compilation of code begins [117]. When looking at the primary functionality of the file system disk quotas is not seen as separate concern, which makes it difficult to configure it when seen as a crosscutting concern. The reasons behind this are: it is difficult to comprehensively reason about quota and identify the structural relationships that it holds, directives can obscure reading of the file system code quota due to scattered code and drift can occur between portions of quota code that should be identical [68, p. 55].

Device Blocking in Drivers

Driver code can be an issue because is the result of multiple independent developers interacting with subtle OS specific protocols though maybe simple to state but hard to manually apply in their scattered and tangled implementation. To make it more complicated, extensions to the scheduling policy of an OS, such as the event-based scheme in Bossa [118], necessarily involve invasive, non-modular, modifications to a rapidly growing number of points in the system to detect events such as blocking. Bossa is a kernel-level event-based framework that facilitates the implementation and integration of new scheduling policies [119] based on a domain specific language approach [120].

2. General Improvements using Aspects

The summary of the results of this paper are shown in Table 2 [68, p. 56] and illustrate the major differences between the original and aspect-oriented implementations of these concerns involve four key related properties: changeability, configurability, redundancy and extensibility. These properties are discussed below in more detail.

Table 4 Summary of the results

Concern	Major evolution	Structural challenge	Original/Aspect	Benefits
Page daemon wakeup	Revamping of code it crosscuts: VM	Multiple context specific thresholds	Scattered activation / Textually localized	Independent development Localized change

	and buffer cache			
Pre-fetching for mapped files	Change in design of sequential mode	New subsystem interaction along execution paths	Internal to function / Explicit control flow	Explicit subsystem interaction pluggability in makefile
Disk quota	New functionality in code it crosscuts: UFS,FFS,EXT2	Configurability And sharing across file systems	#ifdefs w/ redundant code / Explicit sharing	Pointcut configurability Reduced redundancy
Device blocking	New device drivers added to the system	Consistency across rapidly growing diversity	Individualized devices / Centralized assessment	Comprehensive coverage Further extensibility modularized

Changeability

There are two kinds of change, as shown on Table 4 [68, p. 56]. The first type of change is directly to the concern itself which in aspect-oriented implementation was facilitated by textual locality. The second type is indirectly, as a result of revising the code the concern crosscut which should be equally accessible, given tool support. Unfortunately this sort of tool doesn't exist in AspectC yet. Textual locality could also address two further problems in the original implementation. First it could reduce the inconsistencies that arose from non-uniform evolution of the underlying primary modularity the concerns crosscut and secondly putting in one module all the diverse context-specific elements, such as the thresholds in daemon activation. This would create a more natural setting for the original implementation and would enable the developer the spot the differences easier. [68, p. 56]

Configurability

Configuration changes mapped directly to modifications to pointcuts and/or make file options had particular impact on the evolution of both pre-fetching and disk quotas. "Pluggability" is very important for both concerns. In terms of pre-fetching the optimization for sequential mode pre-fetching introduced a new interaction between multiple subsystems and this interaction was unique to a single file system. Explicit configuration as an aspect supported independent development and the eventual removal from the system in case it was needed as it happened with the sequential mode from versions 3 to 4. Similarly in terms of disk quotas aspect the pointcut declarations reveal

the underlying structural relationships between corresponding file system operations. This helps to identify which core file system functions and values are involved, along with their similarities and differences with respect to quota. [68, p. 56]

Redundancy

The elimination of redundancy across file systems increases the configurability of the quota aspect. However there are differences between the implementation of quota in FFS versus EXT2 which cause drift. Therefore the ability to specify similar quota across all file systems eliminates redundant code prevents drift and ensures that quota operations are consistently applied through the system. [68, p. 56]

Extensibility

Scheduling code spans interrupt handlers, device drivers, and all places in the system where process synchronization occurs. One of the challenges in the development of Bossa is to identify throughout the OS all the scheduling points, or the circumstances under which the scheduler is activated. Extending the scheduler to respond to Bossa defined scheduling events requires access to the context of the scheduler invocation, which means invasive modifications to hundreds of places in the system, compromising the modularity of the extension. Aligning the extension as a scheduling concern structured within an aspect could improve the modularity of the extension. [68, p. 57]

3. Runtime Costs

In terms of cost analysis associated with runtime support for the aspect-oriented implementation the constructs of AspectC are static and are resolved at compile time. The current implementation does not introduce more overhead than a call to a function containing the advice body. But cflow is a dynamic construct and hence has runtime overhead associated with it. For cflow construct AspectJ implementation model was followed i.e. which the overhead is distributed across executions of functions that are cflow-tested, and dispatch to advice involving a cflow test. Though AspectC is modelled after AspectJ, there are important differences that still must be addressed such as different kinds of runtime support than is required for user-level AOP etc. [68, p. 57]

Conclusion and Open Issues

AOP proposes new mechanisms to enable the modular implementation of crosscutting concerns. The results thus far have shown that AOP could improve the evolvability of OS code. However, there some open issues that limit this study. [68, p. 58]

1. The focus was only on the evolution of specific concerns in isolation rather than producing full successive versions of FreeBSD.
2. The concerns were evolved by a single developer for all versions.
3. Further aspects could have been considered such as system profiling and networking concerns.
4. An in depth cost/benefit analysis is required because improving modularity of operating systems will not be meaningful if aspects substantially reduce performance.
5. Determine precise costs associated with more sophisticated compositions of aspects in terms relative to their current implementation.

4.3 Case Study II: Persistence as an Aspect

This case study has been adapted from [80] was chosen because persistence is a very relative issue when designing applications. Many of the cited sources were found in the original research but there some critical analysis done regarding the implementations approach that the research [80] suggested when dealing with persistence. Also, the sample source codes are taken from the research but further comments have been added, as it hoped to illustrate the AOP implementation approach in practice for non-trivial applications.

Persistence is considered a classic example for becoming an aspect [121], [122]. Other known examples are synchronisation [123], [124] and tracing [125], [34]. The study [80] claims that persistence can be modularised and re-used using AOP techniques

based on the criteria of Parnas [35]. In addition, applications can be developed unaware of the persistent nature of the data. The result of the study is an attempt to establish evidence of these claims in non-trivial database management systems.

There is currently some research on AOP regarding persistence and related concerns by [126, On to Aspect Persistence], [127, Weaving Aspects in a Persistent Environment] for example, describe an approach and a prototype to store aspects in an O-O database. Therefore, it is imperative to define the main purpose of this study and what is not considered or covered.

- Provides a model for aspect persistence including the persistence of application data, independent of a particular AOP approach.
- Investigates aspectisation of transactions which are only one facet of persistence.
- The transactions considered operate in a pure object-oriented environment, a small share of what is used in the industry.
- Re-factor an existing application.
- Present experiences in separating persistence of application data using AOP techniques.
- Explore whether persistence can be effectively aspectised in a real world application.
- Determine whether such aspectisation can be reusable with the application and the persistence aspect developed independently of each other.
- Provide some general insight into the suitability of other AOP techniques in this context.
- Discuss how the emerging persistence model may be adapted to suit other database technologies, e.g. O-O databases.
- Does not consider the separation of persistence in relational database applications.
- Code modularisation dealing with storage and retrieval of application data from persistent storage is not dealt with in detail.
- Does not explore application development independent of persistence requirements or development of a reusable persistence aspect.
- Does not explore application development independent of persistence requirements or development of a reusable persistence aspect

The basis of the experiment is a database application (a bibliography system) and SQL-92 compliant relational databases as the underlying persistence mechanism. The application is written in Java using Java Database Connectivity (JDBC) and aspectised using AspectJ1.06 [39].

The research [80] uses a classical database application to show that persistence can be a highly re-usable aspect and be developed into a general aspect-based persistence framework. Furthermore, it shows that persistence has to be considered when designing the architecture of data-consumer components where such components need to account the declarative nature of retrieval mechanisms used by many database systems and deletion operation during application design because is highly triggered by most applications.

The approach that was taken to review this paper is a brief analysis on the approach to modularising persistence using aspects and the reason behind the various design decisions including discussing the lessons learnt from the study, possible limitations and generalisation to other persistence scenarios. Related work is discussed when seem fit. As already mentioned it is not indented to analyse any case study in detail but to try to depict the most salient points in order to quantify and assess the claims of AOP and where possible to present a brief overview of the non-trivial application.

Modularising Persistence

1. Database access

When developing aspectised database access, at least partly independent of persistence, it is imperative to consider a way to distinguish persistent data from transient data, while ensuring that the aspectised database access functionality has a high degree of reusability including the availability of some customisation points to plug-in application requirements. Examples of application requirements are a specific database management

system and/or drive, location of the database, points in the application control flow where a database connection should be established or closed.

PersistentRoot class is used to separate persistent data and the concept is taken from O-O database systems [128] whereby is required that all classes whose instances are to be stored in the database extend a common base class. The base class has typically persistence-related functionality and further functionality can be given to the persistent classes by a (pre or post) compilation processor. The PersistentRoot class encapsulates the “marking an object as deleted”, a basic but important feature that allows to be partially ignored during application development. Furthermore, it has an important role in aspectising database access in a highly reusable fashion by the ability to define join points with reference to a common, application independent point: the PersistentRoot class. This allows re-using the DatabaseAccess aspect in other applications whose data classes have been declared as subclasses of the PersistentRoot class. It worth mentioning that an application specific aspect can use AspectJ to declare the PersistentRoot class, which inherits from Object, as the superclass of all classes whose instances are to be made persistent.

The below code extract shows the PersistentRoot class:

```
public class PersistentRoot
{
    protected boolean isDeleted = false;
    public void delete() { this.isDeleted = true; }
    public boolean isDeleted() { return this.isDeleted; }
}
```

The key features of the DatabaseAccess aspect are briefly discussed followed by a commented code of the aspect in order to see a real world example of another application that uses AOP (AspectJ) to modularize crosscutting concerns. Detailed results and analysis can be found in [80, pp. 2-6]:

1. Connection: The ability to connect and disconnect from the database is a basic feature for a persistent application and reusability requirements is required to remain generic with the availability of specific customisation points to incorporate application specific requirements (Examples already mentioned).

2. Storage and update: An object should be stored in the database as soon as it is instantiated. Factors when considering aspectising this functionality: (1) all objects reachable from a stored object should also be made persistent; (2) the constructor must be executed before storing the object. The implementation showed that is essential to treat advices as first class entities in order to clarify the signature of the behaviour specified within an aspect. The declaration of exceptions thrown from the advice code should be incorporated and more reflective access supported which is fundamental in the development of reusable aspects. The update mechanism relies on trapping all invocations of setter methods (calls in its control flow) for persistent objects. It has been decided to rely on strict encapsulation for access to member variables of persistent objects. The research [80] suggests that this practice should be the case all persistent applications as it will ensure that the interface of the class is not modified often due to changes to internal representation of member variables (there are some exceptions).
3. Retrieval information from storage: The application cannot oversee the fact that the persistent objects or the references to these are obtained from an external source that is governed by the declarative nature of retrieval mechanisms in database systems which retrieve data based on predicates or selection conditions. It is interesting to see that aspects can play an important role in modularising parts of the retrieval related code (PersistentData interface). The implementation approach remains application independent and provides a high degree of reusability. Retrieval is an important architectural consideration in the design of data consumer components because there are important factors to consider such as the amount of data.
4. Deletion of persistent data is similar to retrieval functionality in terms of need to be explicitly considered during application development and cannot be fully aspectised. This is because a specific request from the application must be made for the data to be deleted. Because the application is written in OOP, the automatic garbage collection can create uncertainty. Therefore, the paper suggests that is to explicitly

delete persistent objects to ensure that there is no reference on which the aspect can operate. Also, the deletion functionality is reusable and application independent by the use of delete() method as a reference point. The developer does not need to be aware of the existence of the deletion functionality in the DatabaseAccess aspect or the SQLTranslation aspect.

5. Transactions: In brief, the transaction functionality encapsulates the update, retrieve and transactionWrapper methods. Transactions are always implicitly started regardless of the explicit notion of transaction commit offered by JDBC. The advices within the DatabaseAccess aspect do not invoke directly the update or the retrieve method. Rather they pass the name of the method to be invoked together with an array of arguments to the transactionWrapper method which is responsible for catching SQL exceptions during the invocation of the commit and rollback methods and reflectively invoke the required method to decide whether to commit the transaction or rollback. The case study choose to abort a transaction when any exception because it is safer. Also the application does not need to signal exceptions to abort transactions because they are signalled by the aspectisation infrastructure (JDBC, SQLTranslation aspect or Java Reflection API). A similar method is used in [129] with the difference that the transactionWrapper is triggered strictly for database operations and wrapping overheads for transient operations are avoided. As it can be seen, the transactions do not operate in a pure OO environment which in this case benefits the case study design. However, it introduces some overhead to the transactions which is eased by locking optimisation is provided by the update and retrieve methods which establish the appropriate read-write and read-only locks respectively.
6. Meta-data Access: This static inner aspect encapsulates helper functionality, required by the SQLTranslation aspect, to access the database meta-data, for example the column names in a relational Table or its foreign key links. Its purpose is avoid unnecessary duplication of JDBC meta-data calls during SQL translation and built on top of more primitive features available any desired meta-data access feature that is

not supported by the underlying database driver. It is important to mention that this functionality is a subset of the overall database access functionality and because is as an inner aspect of the DatabaseAccess aspect a more natural separation of concerns occurs than it being encapsulated in a sub-aspect. Not to mention that it does not require any concrete or override of features.

The below code segment is the DatabaseAccess aspect with comments:

```
// 1) Connection: No connection pooling implemented, JDBC ODBC driver is chosen which
//offers the lowest common denominator in terms of supported functionality so aspect is more
//re-usable

public abstract aspect DatabaseAccess {

    // variables used to hold the connection information
    private static Connection dbconnection;
    private static String dbURL;

    // To obtain information to connect to the database abstract methods are invoked by a before
    //advice operating on the abstract pointcut establishConnection()

    abstract pointcut establishConnection();
    abstract pointcut closeConnection();

    //DB URL & Driver Details are supplied by an application aspect extending the DatabaseAccess
    //aspect
    public abstract String getDatabaseURL();
    public abstract String getDriverName();

    // 2) Storage and update: Object should be stored in the database as soon as it is instantiated
    //(after its constructor has been executed) and all objects reachable from it should also be made
    //persistent, the objects are written to the database through translation to SQL insert statements
    // Pointcuts identify the join points where an object should be stored in the database or its
    //persistent representation updated. Update mechanism relies on trapping all invocations of
    //setter methods for persistent objects. Method is used to rebuild the objects from their relational
    //representation

    pointcut trapInstantiations(): call(PersistentRoot+.new(..));
    pointcut trapUpdates(PersistentRoot obj):
    !cflow(call(public static Vector SQLTranslation.getObjects(resultSet, String))) &&
        (this(obj) &&
        execution(public void PersistentRoot+.set*(..))
    );
}
```

*// 3) Retrieval. The interface is used to provide hooks by trapRetrievals pointcut to identify the
//points at which the application tries to retrieve the data. All these methods return a Vector
//containing the objects retrieved.*

```
pointcut trapRetrievals( ):
    call(vector PersistentData.get*(..));
```

*// provides a reference to an instance of a class implementing this interface where an application
//can obtain this reference and use it as the basis of any retrieval-related code*

```
public static PersistentData getPersistentData() { ... }
```

*// 4. Deletion Application invokes this method for the persistent instances. The trapDeletes()
//pointcut captures these invocations and a before advice, translates the request to SQL using
//the SQLTranslation aspect & removes the persistent representation of the object.*

```
pointcut trapDeletes(PersistentRoot obj): this(obj) &&
    execution(public void PersistentRoot+.delete());
```

*// The detectDeletedObjects pointcut complements the trapDeletes() pointcut by throwing an
//exception (wrapped as an AspectJ SoftException) whenever the application tries to access the
//transient representation of a deleted persistent object that has not yet been collected by the
//garbage collector.*

```
pointcut detectoeletedobjects(PersistentRoot obj): this(obj)
    (execution(public * PersistentRoot+.get*(..)) ||
    execution(public * PersistentRoot+.set*(..)) ||
    execution(public string PersistentRoot+.toString()))
    );
```

*// 5) Transactions: the update and retrieve methods encapsulate the code that results in the start
//of read-write and read-only transactions respectively. className argument, for all the methods
//in the PersistentData interface, is obtained by the advice operating on the trapRetrievals
//pointcut. className establishes the mapping between the object structure and the underlying
//relational schema.*

```
protected static Integer update(string sqlStatement) throws SQLException { ... }
protected static Vector retrieve(string sqlStatement, string className) throws
    SQLException { ... }
protected static object transactionWrapper(string methodName, object[] params) { ... }
```

*//6 Meta-data Access: encapsulates helper functionality, required by the SQLTranslation aspect,
//to access the database meta-data.*

```
public static aspect MetaDataAccess { ... }
```

```
// advice code
}
```



2. SQL Translation

As shown earlier database access is a concern for any application persistent data. But this may not be the case for translation to the underlying model. Therefore, it must be considered as a separate concern when aspectising persistence of OO data using relational databases. Also due to the lack of support for complex data types in relational database the object structure must be flatten so that the inheritance relationship is captured by a simple one-to-one relationship and with an additional relational Table to accommodate many-to-many relationship.

The SQL translation provides the object-to-relational mapping that is required for this application [80, pp. 6-7]. It important to mention that JDBC ResultSet objects were considered to be employed in order to modify the database instead, but unfortunately not all JDBC drivers support use of bi-directional cursors on result sets, an imperative requirement to search for records. This approach requires retrieving the object into a ResultSet and applying the update which results in unnecessary disk access. However, pure relational databases are not supported by the SQLData interface in JDBC because they only support mapping to or from user-defined SQL types in an object-relational model [80, p. 6].

The case study has taken the approach of a singleton lookup Table to establish the mapping. This approach was taken so it can be reusable and independent of application-specific mapping. The use of the lookup Table is further minimised by maintaining a broader granularity mapping (the Tables to which objects of a class and many-to-many relationships map). EstablishMapping aspect specifies the mapping of the lookup Table which, sets up the mapping before the connection with the database is established. Also, EstablishMapping aspect should have a higher execution priority than the DatabaseAccess aspect so that the mapping is established before connecting to the database.

The SQLTranslation aspect main features are shown the below code segment with added comments. While inspecting the code segment it important to mention that the mapping to multiple SQL statements is an SQL translation concern. Therefore, the pointcut

dealing with this must form part of the corresponding aspect. Also, in order to maintain good AOP practices it is important to be able to separate an essential piece of SQL translation functionality and incorporate it within the SQLTranslation aspect. This is possible even if the sqlExecution pointcut captures Statement.executeUpdate(String) calls from a single update method in the DatabaseAccess aspect. [80, p. 6]

*//Normal execution in the DatabaseAccess aspect proceeds when the around advice checks if a
//single SQL statement is being executed through the JDBC Statement object.*

```
public aspect SQLTranslation {
```

*//sqlExecution pointcut is used to capture if an object maps to multiple Tables that would result
//in translation multiple SQL statements, executed in a batch mode.*

```
pointcut sqlExecution(Statement statement,String sqlstatement): target(statement)
    && call(public int Statement.executeUpdate(String))
    && args(sqlstatement);
```

*// around advice for sqlExecution pointcut
// The methods below employ Java Reflection and the mapping information in the lookup Table
//to map the objects, their updates and deletion to the database and recreate the objects upon
//retrieval*

```
public static string getInsertionSQL(PersistentRoot obj);
public static String getupdateSQL(PersistentRoot obj, String methodName, object arg);
public static String getDeletesQL(PersistentRoot obj);
public static string getQuerySQL(String className, String selectioncondition);
public static vector getobjects(Resultset rs, String className);
```

```
// helper methods
}
```

The SQLTranslation aspect needs to be very flexible and, therefore, the case study chose to use the various methods to employ Java Reflection a powerful technique that can enable applications to perform operations which would otherwise be impossible. Reflection is a relatively advanced feature, commonly used by programs which require the ability to examine or modify the runtime behaviour of applications running in the Java virtual machine. [130]. Using this feature the mapping information in the lookup Table to map the objects, their updates and deletion to the database and recreate the objects upon retrieval.

According to [130] reflection is powerful, but should not be used promiscuously. If it is possible to perform an operation without using reflection, then it is preferable to avoid using it. The concerns that have been noted when accessing code via reflection are: (1) Performance overhead, because it involves types that are dynamically resolved that certain Java virtual machine optimizations cannot be performed. Therefore, reflective operations have slower performance than their non-reflective counterparts. (2) Security restrictions, because reflection requires a runtime permission which may not be present when running under a security manager. This is an important consideration for code which has to run in a restricted security context, such as in an Applet. (3) Exposure of internals, since reflection allows code to perform operations that would be illegal in non-reflective code, such as accessing private fields and methods, the use of reflection can result in unexpected side-effects, which may render code dysfunctional and can destroy portability. Reflective code breaks abstractions and therefore may change behaviour with upgrades of the platform. The case study suggests that only additional overhead is caused during database interaction arguing that [131] points out that such trade-offs have to be made when designing highly flexible components such as the SQLTranslation aspect.

Furthermore, because of encapsulation restrictions the object attributes corresponding to the relational Table columns are identified recursively (defined in terms of itself) by obtaining the declared members and not just the public ones. To ensure consistency within a single transaction boundary the linked Tables are updated individually in case the propagation of updates for linked Tables is not supported in the underlying database design.

As mentioned, reflection has played an important role in the design of reusable transaction wrapper and, more importantly, SQL translation mechanism by generalizing wherever application specific code would be required otherwise. The drawbacks explained earlier relate to the SQL translation and hence the well defined assumption that strict encapsulation is enforced (only get/set methods are used for public access to an object's state). If developers ignore this assumption the translation mechanism

method would fail to operate. To overcome this issue support can be given for generating get/set methods or add a declare feature in AspectJ to ensure that developers define these methods. In terms of performance overhead, the suggested solution is cache which becomes an issue as the database grows. At least the pointcuts of the DatabaseAccess aspect can provide reference points for plugging a cache into the persistence model.

The suitability of AspectJ is good for aspectising persistence because is a general concern regardless of the individual state of an object (pointcuts and advices is useful). The relationships in this implementation are done as aspects mainly relying on AspectJ introductions. Unfortunately, because of their complexity in this implementation they introduce additional overhead and therefore must be used very carefully with a well define model such as the one suggested in the case study in the dynamic relationships in OO Databases [132] using composition filters as in [133]. This suggests the need for environments that allow multiple AOP techniques and platforms to co-exist hence allowing the use of the best technique for modularising a particular crosscutting concern.

Another important factor is aspect interactions, these interactions cut across aspects in a system. The case study [80, p. 8] suggests that it imperative that AOP techniques in general offer proper support for the detection, modularisation and resolution of interactions. This is fundamental for testing and verification of aspect-oriented applications and therefore, a critical factor in large scale adoption of aspect-orientation.

3. The Emerging Persistence Framework

This implementation of the DatabaseAccess and SQLTranslation aspects and their results, like the previous case studies show scattering and tangling code can be minimised by modularising crosscutting concerns and thus achieve aspectisation. This may be true for simple cases but as it is shown in Figure 13 aspectisation requires a need for collaboration of coherent set of modules including classes and aspects. [80]. Figure 13 illustrates a general aspect-based persistence framework that emerged from the

discussion in the previous sections. This will allow to work upon well established practices and guidelines from the frameworks community as shown with the example of the case study in terms of flexibility trade-offs. The arrows in the Figure 13 denote usage.

Therefore, AOP ensures that aspectisation leads to a natural separation of concern such as the separation of the DatabaseAccess and SQLTranslation aspects in the case of persistence framework. This is also augmented in [134] which presents metaphor based classification of crosscutting concerns, which is driven by their manifested shapes through a system's modular structure.

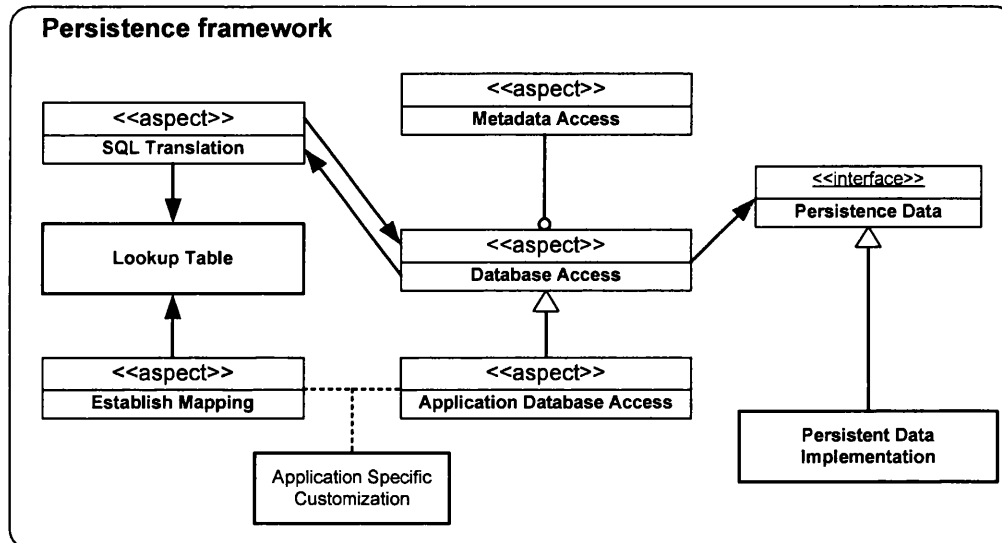


Figure 13 Persistence framework from [80]

In terms of using other persistence mechanisms, as mentioned the persistence framework is from a classical relational database application so any OO application can reuse it by employing an SQL-92 compliant relational database. Note that the framework will need to be re-implemented if OO databases are used. This means that the SQLTranslation aspect, lookup Table and EstablishMapping aspect will not be required as there will be no data model issues between the OO application and the database. Therefore, MetaDataAccess aspect will not be needed either as it is only needed to support SQL translation.

Regardless of the nature of the persistence mechanism, DatabaseAccess aspect will be required as these are the points in the application control flow where persistence features are composed. Also, as mentioned a transaction wrapper will be required and a PersistentData interface to support declarative access from the application. Finally the PersistentRoot class will be required. The approach has worked successfully when designing aspect persistence mechanisms in the past [126, On to Aspect Persistence], [127, Weaving Aspects in a Persistent Environment] where the PersistentRoot class. Finally in terms of reflection, if the resulting persistence framework were to be implemented in another language environment, both the base language and the aspect language would need to support reflection.

Conclusion

The aim of this study was to assess if AOP techniques offer an effective means to modularise persistence in a real world application scenario. The outcome was positive with a number of important software engineering factors to keep in mind.

Firstly, the necessity of the trade-offs between generalization and performance. The application specific statements in the SQLTranslation aspect were not hardcoded like in [135] but used reflection instead. This allowed for generalization and reusability of the SQL translation mechanism i.e. the aspectised persistence mechanism.

Secondly, well modelled aspects require investigation the suitability of the available techniques for implementing the various concerns within the aspect. For example, the use of AspectJ constructs to identify points where persistence-related behaviour has to be composed while reflection has been used to keep the SQL translation generic and avoid duplication of transaction code during database access. However the choice of suiTable technique is limiting the available tools and the way they interact. So instead of using composition filters AspectJ introductions were used.

The study also tried to answer to two questions:

1. Can a persistence aspect be designed so it can be re-usable?

The result was also positive, the answer illustrated a persistence framework that does not rely on the existence of an additional layer masking the relational database features for example the DatabaseAccess aspect. However, the re-use of the framework should be strengthened by re-use of specification which clearly defines the interface of aspects behaviour.

2. Can an application and a persistence aspect be developed independently of each other?

The case study showed that this can be partially. For example storage does not need to be considered but retrieval is essential. The implementation details of the application were not considered so the persistence mechanism had to be generic hence re-usable. All these allowed natural separation of concerns while developing the persistence infrastructure and keeping the reusability and application independence requirements which, resulted in the framework.

Suggestions for further work would include performance concern with non-AO techniques, the suitability of other languages and the implementation of persistence in a real world application.

4.4 Case Study III: AOSD with Use-Cases

Crosscutting concerns are responsible for producing spread and tangled representations throughout the software life cycle. Effective separation of such concerns is essential to improve understandability and maintainability of artefacts at the various software development stages [9], [25, p. 43]. Aspect-oriented software development holds promise for the purpose. There are numbers of papers [138], [139] discussing UML-based realisation approach of the general aspect-oriented requirements engineering process.

While [139] described a viewpoint-based implementation of the process, this case study describes the experience gathered using the Aspect-Oriented software development use-case driven approach. Use-case driven approach provides a sound method for

developing applications by focusing on realizing stakeholder concerns and delivering value to the user. It has been shown so far that aspect orientation has helped with modularizing crosscutting concerns. However, most of the work shown so far in this area has concentrated on the implementation phases [68], [80]. The use-case driven approach attempts to modularize crosscutting concerns much earlier, even during requirements. The underlying concept in aspect orientation is similar to the concept of use-case-driven development.

The study is based on a non-trivial new user provisioning system application adapted from an established company and attempts to outline how to conduct AOSD with use-cases in the requirements and analysis stages of the particular project. The study look at a small subset of the intended solution for the non-trivial application, namely the gas wrapper server replacement. Use-cases will be used to demonstrate the way that separation of crosscutting concerns can be achieved for the user access management processes. The study mainly focuses on the requirements and design of application architecture with particular emphasis given to infrastructure use-case modelling based on Jacobson methodology [140].

4.4.1 Introduction

The company is driven to optimise their application's performance in the global marketplace, hence a programme to transform their ecommerce infrastructure platform was initiated. The programme's purpose is to enable the company to reduce the cost and complexity of delivering changes to the ecommerce estate and deliver a financial benefit that is driven by productivity improvements and avoidance of maintenance costs. The programme is offering a technically complex solution which, as an overall integrated architecture, is not yet proven. The implementation of the new platform will be achieved by replacing the current IBM WebSphere system [141]. The choice of using technologies such as WebSphere was mainly due to shorter application development time, the ability to support up-coming legislative changes and the reduction of costs through web-based automation. One of these technologies to be replaced as part of this assisted transformation programme is the current TAM (Tivoli Access Manager) [142]

security infrastructure. The existing infrastructure is based on obsolete versions of the software and the Gas Wrappers (locally-developed code) which are used for user registration and management. This is causing instability and performance issues.

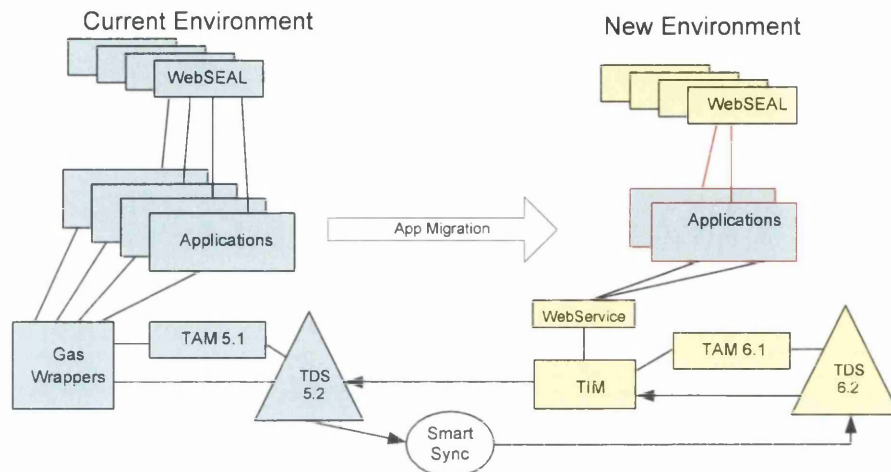


Figure 14 High level view of the current and new environment.

Figure 14 depicts a high level view of the existing and new environment of the area of interest. It shows the architecture of a replacement system which, while continuing the use of TAM, will create a more resilient and flexible infrastructure to provide authentication and access control. The new system will standardise the security layer offering, improving stability and supportability by replacing custom code with TIM (Tivoli Identity Manager) [143]. The solution allows applications to be migrated in phases and ensures that when users register in either environment both registration databases are in sync.

The work for the replacement of the security infrastructure is broken into three distinct sections with the following objectives:

1) Security service infrastructure

This entails the creation of a well-defined security service infrastructure that provides access control and identity management to the WebSphere application environments with added build capabilities to allow rapid and consistent deployment. The solution addresses requirements for fine-grained access control within service-layer applications.

2) TAM upgrade

This deals with the replacement of the TAM and Tivoli Directory Server (TDS) software components from the present obsolete versions. The TAM configuration becomes simplified as part of the upgrade without impacting the application. The design should also allow the simple and repeatable creation of a new environment.

3) Gas Wrapper server replacement

The replacement will provide a new user provisioning system to replace the Gas Wrapper servers and remove the current significant obstruction to the stability and maintainability of the TAM infrastructure. The proposed solution seeks to clarify the interface between applications and the security infrastructure, by separating the provisioning component and the authorization concerns. The proposed provisioning system will be based on TIM which offers a workflow engine and generic administration interface, thereby reducing the amount of custom code required to be written and offering capabilities for easier future expansion.

4.4.2 Solution Architecture

A simplistic way would be to upgrade TAM and TDS and modify the gas wrappers to work with these new versions, however, the locally-developed code that is currently used for user registration and management is causing a number of issues:

1. The technology base on which they are built is obsolete and not strategic for the company.
2. Maintenance and enhancements are problematic because of limited resources in the TAM support team and the necessary skills within. Subsequently, requests for change tend to take a long time to deliver.
3. Tendency to avoid making changes “in case it breaks something”.
4. The web interfaces offered are at a very granular level and are more appropriate for local access than a networked service layer.

5. Although the individual API functions are documented, there has been less guidance regarding the context in which they should be used. This has led to problems with performance and instability in functions that are intended for user provisioning which is used at application login.
6. There is no clear separation of crosscutting concerns in terms of the infrastructure.
7. There is minimal security surrounding the invocation of the Gas Wrapper service.
8. Many applications of various business branches are hosted in the current infrastructure and any changes would impact the entire application stack.

As a result of the above issues there is a lack of consistency and scattered code among the applications that use the services. This means that different applications with similar access requirements may have slightly different behaviour and that within the Gas Wrapper code there are a number of sections that do almost the same thing but with subtle differences. As a result of ad-hoc use of the Gas Wrapper services over time, the TAM user registry has become the authoritative source for some application-specific data that is not directly related to its security role and which would be better placed in a business data repository.

Figure 15 shows a simplified view of the suggested user access control architecture. The architecture shown here would apply to almost all customer-facing applications and to WebSphere-hosted applications that require access. The WebSeal layer operates as a policy enforcement point for access control definitions which are maintained by a central TAM policy server. Internet users will be authenticated using password, PIN and certificate information held in the user registry. A user provisioning service will allow user accounts to be created and for their access entitlements to be set. The rules for provisioning will use policy definitions. The provisioning process will interact with the end user and the applications.

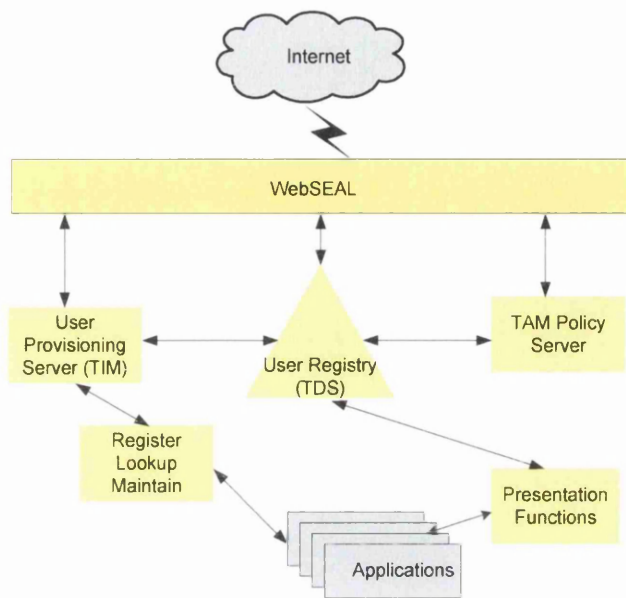


Figure 15 User Access Control Architecture

The access control components could be divided into two distinct parts: the Access Control Layer (TAM WebSEAL) and the Provisioning System (TIM) as shown in Figure 16. To gain access to the business application, the end user must be allowed access by the Access Control Layer. The user's interactions with the access control layer include such activities as logging in, changing passwords and receiving error messages. These are provided by the Access Layer Presentation Function.

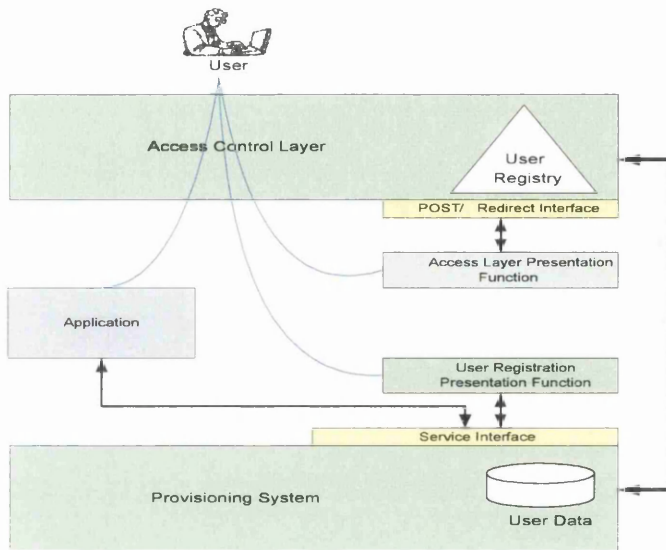


Figure 16 User Access Control showing Presentation Layer Components

The processes for registering, updating and setting access entitlements for users are implemented by the Provisioning System, the core of the identity management solution. Its role is to handle the creation and maintenance of users. This is done by maintaining a database of all the users in the system together with their access rights. It would then apply policy rules to determine the accounts to create and entitlements to assign to them within the Access Control Layer. The role of the Access Control Layer is to enforce access policy. The Access Layer Presentation Function handles all interactions between the user and the Access Control Layer and also provides pages that interact with the Provisioning System when the Access Layer intercepts particular situations during user registration. The Registration Presentation Function provides users with a self-registration and self-service interface. It is based on Web Services exposed by the Presentation System.

For every access request, the Access Control Layer should perform two checks:

1. Is the user a member of a group that is permitted to access the requested resource?
- 1) Has the user been authenticated at an appropriate level for the protected resource?

The first check is a conventional role-based access control. When an application is defined to the security service the URL patterns to be protected and the access groups to be given access will be defined in the form of access control lists. The second check is drawn from the company Group Standards and defines access and the authentication for each of them.

The Application must also have pre-registered one or more Access Definitions with the Provisioning System that can act as an index into a metadata table, containing some, or all of the following:

1. Access Permissions
2. The Access Layer group(s) to which registration with the Access Definition should give access
3. The name of the Registration Process required for the Access Definition
4. The name(s) of any other Registration Process(es) that are trusted by the Access Definition
5. The initial URL for the application

4.4.3 Capturing Concerns with Use-Cases

4.4.3.1 Requirements Gathering

In order for the above suggested architecture to be built properly it is important to understand the stakeholders' real concerns. Understanding these concerns is critical to successful software development and to build the correct system it is imperative that the requirements have been properly captured. In addition to this, it is important to understand stakeholder priorities as not all concerns are of equal importance. The priority determines which requirements have to be developed before others so that if things do not turn out well some requirements can be dropped and your stakeholders still get an acceptable, albeit incomplete, functionality. Stakeholders normally justify the need for a new system or an enhancement by emphasizing the benefits and payoffs of

specific features. Features would be high-level statements of desired capability; these can either be in terms of functionality (i.e. what the system can do) or some other quality attribute (performance security etc). The listing below is a subset of the key requirements that the system should be able to perform.

Table 5 Functional Requirements

Ref	High level description	Detailed description
FU1.1	Online Account Creation	The ability to create an online account as part of the registration process. (Has the required authorization). The account should have the following attributes: Userid, Password, PIN and E-mail.
FU1.2	Online Account activation	The ability to support the activation of an account after the creation of that account.
FU1.3	Query Account	The ability to query an online account for status, group & role membership and the last logged on date & time.
FU1.4	Online Account Maintenance	The ability to programmatically: Change Userid Change Password Change PIN
FU1.5	Online Account Forgotten Details support	The ability to programmatically: Retrieve Userid Reset Password Reset PIN

Table 6 Non-Functional Requirements

Ref	High level description	Detailed description
NF2.1	Performance – Application response time	All transactions should take no longer than 3 seconds.
NF2.2	Authorization	All transaction must have the necessary permissions
NF2.3	Application scalability	The ability to support up to 1000 online account registrations per day
NF2.4	Smart Synchronization	The ability to synchronize both environments when registration occurs

One way to find out more about these requirements is to refine them, one by one, resulting in a long list of requirements which may end up with loose pieces of information. A more effective method is to walk through the use of the system and uncover how the features are put into effect. This method puts features in context of the system operation.

4.4.3.2 Use-Case modelling

In the earlier case studies it has been shown that an extensible system can be achieved by keeping concerns separate all the way to the code and modularize the implementation accordingly, however it was not shown in detail how to find concerns and express them clearly. Using the use case technique it is hoped to explore the various ways in which a system is used validating the stakeholders concern early in the project and drives the definition of the system architecture [138, p. 29]. As the focus of the study is on infrastructure, use-case modelling the above requirements will be depicted from an architecture components perspective. Most practitioners model functional requirements with use-cases but they tend to leave non-functional requirements out of use-case modelling. However, as long as a requirement requires some observable response to be programmed into the system, use-cases can be applied. In the case of non-functional requirements, as they usually need the support of some underlying infrastructure mechanisms, they are therefore called “infrastructure use-cases”. All functional concerns are depicted through application use cases [138, p. 85].

Figure 17 is a first attempt at identifying use-cases that address the above requirements. It describes some of the use-cases for customer interactions on which the design is based. These use-cases are only a subset of the possible ones, with the intention of indicating the processes for some key interactions between customers, applications, and the provisioning and access control functions. The self-registration process has three peer use-cases. Peer use-cases are those which have no relationship between them. They are distinct and separate but their realizations overlap and they impose responsibilities on the same classes [138, p. 40].

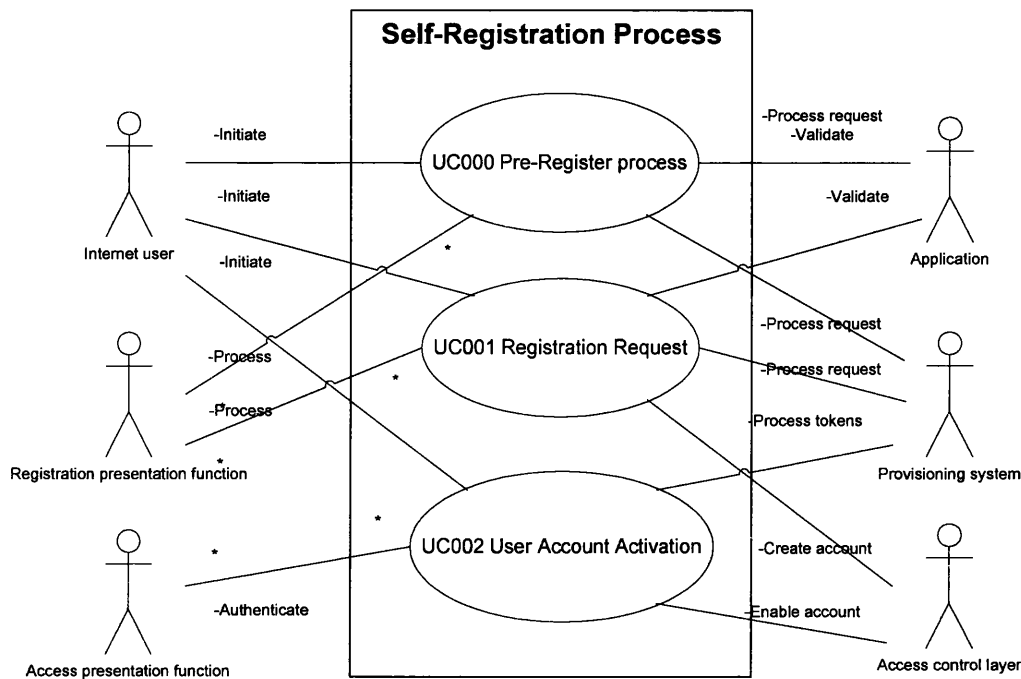


Figure 17 Self-registration process use cases.

The table below depicts the actors involved in the use-case modelling for this section of the solution design.

Table 7 Actor names and their description

Actor Name	Description
Internet User	A user who will be accessing data as specified in the Company Group Customer Authentication standard. They will authenticate to the system with a user name, password, and self-chosen PIN.
Application	The application to which the internet user is seeking to gain access.
Provisioning System	This is responsible for maintaining the information that allows a user to register for, and gain access to, applications. In this design the role is taken by Tivoli Identity Manager (TIM).
Registration Presentation Function	This is the presentation layer code for the Provisioning System that interacts with the user during user-registration and self-care operations.
Access Control Layer	This provides a gateway for controlling access to the applications. In this environment this role is taken by Tivoli Access Manager (TAM) WebSEAL.
Access Presentation Function	This is the Presentation Layer code that interacts with the user during access control operations. It can be thought of as the presentation part of the Access Control Layer.

The use-cases have now been identified for the system and, in effect, the different concerns for a system have been separated from the actors' perspective. The next step is to explore the concerns of use-case in greater detail and then, for each use-case, identify the flow of events describing how a particular variation is handled through that use-case. Each use-case consists of a basic flow (main scenario) and all variations are described as

separate alternative flows (alternate scenario) to prevent the basic flow being entangled by all the variations that the use-case needs to handle [138, p. 54]. The Pre-Registration Process, Registration Request and User Account Activation use-case specifications have been selected for the purposes of this study.

4.4.3.3 Use-Case Specification

Table 8 Self-Registration process – Use-Case 000 specification: Pre-Registration Process

Name	UC000 – Pre-Registration Process
Description	In this use case, the Application establishes a new user's entitlement for access.
Pre-Conditions	Main scenario: a page within the application will be registered in the access layer checks if the user has the required authorization that is required by the Application.
Post-Conditions	N/A
Main Scenario	
Step	Actions
1	The Use-Case starts with a request from the Internet User to the Application to register a new account.
2	The Application prompts the Internet User for personal and policy details. The application performs standard processing to validate that the user is a recognised customer and that they are entitled to access the application.
3	If the Application elects not to permit existing users to be given access to the application, the use-case ends at this point.
4	The Application requests a search in the Provisioning system for any user whose registered email address or login name matches the email address provided by the user. If no matching accounts are found, the use-case ends.
5	The Application checks the response from the Provisioning system to see if any of the returned accounts has a registration process that has been initiated but not yet completed (a user should not have more than one registration process outstanding). If this is the case go to [AS00001 Terminate duplicate registration request]
6	The Application informs the Internet User if conflicting account(s) exist. If accounts are conflicting it then gives the user a choice of adding extra access to an existing account or continuing. If the user chooses to create a new account, the use case ends.
7	If the user continues using an existing account name the Registration Presentation Function checks to see if the chosen account has the authorization that is required by the Application. If this is not the case go to [AS00002 Receive the required Authorization], otherwise it redirects the Internet User to a URL protected at the same level as the user's access level and the user is prompted to login.
8	The Application calls the Provisioning System to assign its AccessDefinition to the user.
9	The use-case ends
Alternative Scenarios	
Step	Actions
AS00001 Terminate duplicate registration request	
1	At step 2 in the main scenario, the Application determined that an account registration request was already in progress for the user.
2	The Application asks the Internet User whether the existing request should be aborted. If confirmed, the Application calls the Provisioning System to delete the inactive user.
3	The Use-Case ends.

AS00002 Receive the required Authorization	
1	At step 4 in AS00001, the Application determined that a user has not the required authorization by the Application.
2	The Application redirects the user to a page which requests security questions. If successful it then calls the Provisioning System to define the required permissions and update its AccessDefinition that will be assigned to the user account.
3	The Use-Case ends.

Table 9 Self-Registration process - Use Case 001 specification: Registration Request

Name	UC001 – Registration Request
Description	This Use-Case initiates a user self-registration process. In it, the user's entitlement to access is validated, user information is captured and user identity and registration tokens are generated.
Pre-Conditions	The application must have established the user's right to hold a login account by following Use-Case UC000.
Post-Conditions	The user's entitlement for access has been established. A pending registration process is active in the Provisioning System. A new, as yet unregistered, user has been added to the Access Control Layer user registry. Activation token(s) have been sent to the new user.
Main Scenario	
Step	Actions
1	The Use-Case starts when a user initiates the self-registration process and the Application calls the Provisioning System to initiate a registration process. The Application passes the Access Definition ID, user's name and email address to allow the registration to proceed and then passes control to the Registration Presentation Function.
2	The Provisioning System performs the following sequence of operations: 1) Generate a unique user account ID for the user 2) Initiate an Add operation to store the application-provided data together with the user account ID as a new Person in ITIM. 3) Respond to the Application returning a registration handle containing the user account and the request ID of the Add operation.
3	1) The Application redirects the user to the Registration Presentation Function and requests the ID received by the registration response. 2) Register the user [AS00101All-in-one User Registration]
4	The Registration Presentation Function calls the Provisioning System to retrieve Access Definition data.
5	The Registration Presentation Function presents a form to the Internet User containing: 1) User ID 2) Initial Password plus verification field. 3) Password recovery questions.
6	The Registration Presentation Function calls the Provisioning System to continue the registration process. The request includes the registration handle, chosen user ID and chosen password.
7	1) The Provisioning System validates that the user ID is unique and the password is acceptable to the standard password policy. If not, it responds negatively to the caller and the user is requested to try again. 2) The Provisioning System initiates a Modify operation for the user which will assign them an account in the Access Control system with the appropriate permissions based on the Access Definition and with a flag to indicate that registration is in progress. 3) The Provisioning System responds positively to the caller.

8	The Provisioning System generates a random activation code and PIN which it sends to the Internet User in an email containing a link to an activation URL.
9	The Use-Case ends.
Alternative Scenarios	
Step	Actions
AS00101 All-in-one User Registration	
1	At step 3 of the Main Scenario, the Application collects all required information to generate a new user definition instead of passing the collection of user name, password etc. to the Registration presentation function. In this case, the application passes the following data to allow the registration to proceed: Access Definition ID User's name User's email address Users's chosen ID
2	Processing continues at step 7 of the Main Scenario

Table 10 Self-Registration process – Use-Case 002 specification: User Account Activation

Name	UC002 User Account Activation
Description	This use-case completes a user self-registration process initiated by UC001. In it the user presents the activation code(s) that the Provisioning System previously generated during UC001. These are checked and the user registration details are updated if the check is successful.
Pre-Conditions	UC001 must have been completed for the given user and application ID.
Post-Conditions	All scenarios: <ul style="list-style-type: none"> The user has been given access to the application The user has accessed the application for the first time
Main Scenario	
Step	Actions
1	The Use-Case starts with a request from the Internet use to access a URL in the Registration Presentation Function. This will happen as a result of following a link in the email they received containing the activation token and will be to the special activation URL within the Access Presentation Function.
2	The Access Presentation Function presents the user with a form in which to enter the following details: <ol style="list-style-type: none"> 1) User name 2) Password 3) The activation code from the activation email. 4) The initial PIN code from their activation email.
3	The Access Presentation Function checks the user name, password and other token(s) provided by the user. If any of them are incorrect an error message is presented. If the username is recognized the Access Presentation Function will increment a counter and compare it with the maximum retries defined in the Access Definition. [AS00202 Credentials entered incorrectly too many times]
4	When the user has successfully entered their credentials, the Access Presentation Function calls the Provisioning System to flag the user as “no longer pending registration” and the user is prompted to change their PIN. [AS00201. Update PIN]
5	The Access Presentation Function then presents the user with a confirmation page which contains a link to pass them to the application home page.
6	When the user clicks the link the Access presentation Function redirects the user to the initial URL of the Application (drawn from the Access Definition).

7	The Use-Case ends.
Alternative Scenarios	
Step	Actions
AS00201, Update PIN	
1	At step 4 of the Main Scenario, the Access Presentation Function determines that the activation is in progress
2	The Access Presentation Function presents the user with a form requesting them to change their PIN.
3	The use-case continues at step 6 of the main scenario.
AS00202 Credentials entered incorrectly too many times	
1	At step 3 of the Main Scenario, the Access Presentation Function determined that a user presented for activation had failed to enter their password, PIN or activation code(s) correctly.
2	The Access Presentation Function calls the Provisioning System to suspend the user. It then presents the user with a page informing them that their account has been locked-out.
3	The Use-Case ends.

4.4.3.4 Use-case Slices

Use-cases provide the means to model and separate crosscutting concerns effectively and is important that this is preserved through design and implementation. This can be achieved by collating the specifics of a use-case during design in a modularity unit known as a use-case slice [138, p. 36]. Each use-case slice collates sections of classes, operations, and so forth, which are specific to a use-case in a model. From this perspective, the tangling of concerns is avoided and parallel development and managing system configuration is assisted [138, p. 37].

Table 11 Composing peer use-case realizations with use-case slices.

Use-Cases	Extensions of behaviour specific to use-case realization					
UC000 Pre-Registration Process	x	x	x	x		
UC001 Registration Request	x	x	x	x	x	
UC002 User Account Activation	x		x		x	x
	Internet User	Application	Provisioning System	Registration Presentation Function	Access Control Layer	Access Presentation Function

From the above use-cases and use-case specifications, Table 11 depicts all specifics to a use-case slice, therefore each use-case slice may not have complete classes but have part of classes (class extensions). In essence, these contain only the features of a class needed to realize a specific use-case. It is worth noting in Table 11 that each horizontal row shows a use-case slice containing the extensions of classes needed to realize the use-case [138, p. 41].

4.4.3.5 Visualizing Use-Case Flows

An alternative way of visualizing the flow of use-cases is to depict them in compartments. The ellipse notation within the top compartment depicts the use of use-case. UML [144] allows tags to define values and give more information about particular elements in the model. The tags {basic}, {alt} and {sub} are not defined in UML but introduced by Jacobson et al. [138, p. 58]. The tag {basic} indicates that a flow can be triggered by an actor. The {alt} indicates an alternate flow triggered by an actor or application instead of the basic main flow. The tag {sub} indicates that a flow can be referenced or included only by another flow. Inclusions and extensions are opposites. With extensions, an extension flow inserts itself into the existing use-case flow, whereas with inclusions, it is the responsibility of the existing use-case flow to insert the inclusion flow.

An extension use-case flow is realized by an advice [138, p. 43]. The extension points in use-cases correspond to the points in the execution flow in AOP i.e. join points. Pointcuts can refer to multiple extension points (join points) that may be defined in multiple classes at once. This is advantageous especially for infrastructure mechanisms such as authorization, performance etc. Figures 18, 19 and 20 visualize the selected use-case specification including the extension pointcuts.

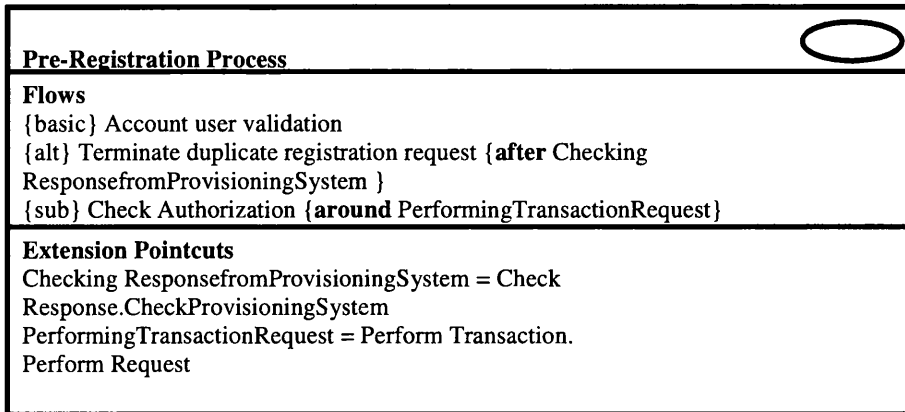


Figure 18 Pre-Registration Process

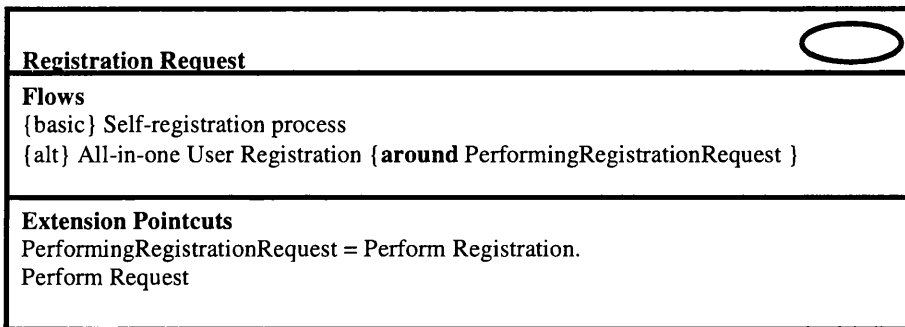


Figure 19 Registration Request

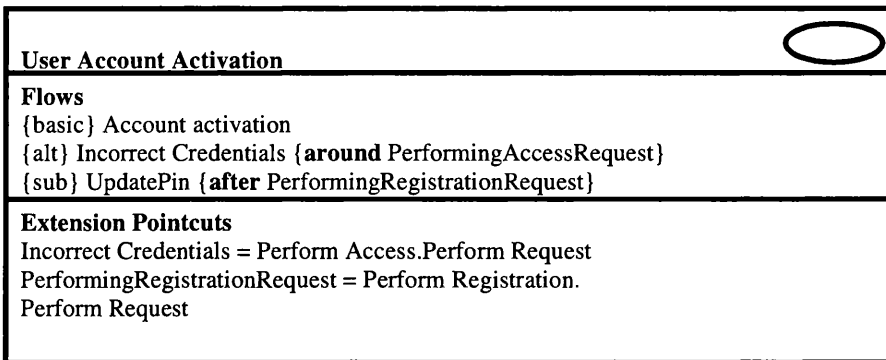


Figure 20 User Account Activation

4.4.3.6 Capturing Infrastructure Use-Cases

The non-functional requirements mentioned earlier such as authorization and synchronization can be refined and kept separate as infrastructure use-cases and modelled as extensions to application use-cases. There are also other kinds of non-functional requirements that deal with system wide qualities such as performance and scalability. These system wide concerns are described simply as declarative statements during requirements. There are usually several key infrastructure use-cases that are used to achieve these qualities and the sum of these infrastructure uses-cases need to be considered in order to determine whether these qualities are met [138, p. 93].

Table 2 non-functional requirements are qualities of the system that are required for each step of an application use-case. Each step of use-case is called a use-case transaction. It is an actor request-system response pair; the actor does something, the system responds in return. Since the requirements need additional processing within the basic use-case transaction, the non-functional requirements can be modelled as extensions to this basic transaction. The basic transaction can be modelled through a <Perform Transaction> use-case as shown in Figure 21 [138, p. 94].

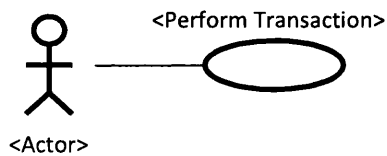


Figure 21 <Perform Transaction> use-case

The <Perform Transaction> use-case is very important to the architect. It is from these use-cases where infrastructure mechanisms are introduced. During analysis, design and implementation the realization of <Perform Transaction> use-case becomes a pattern that is applied to the realization of each application use-case step. For systems with more infrastructure concerns, different extension use-cases can represent a separate non-functional concern. This is shown in Figure 22.

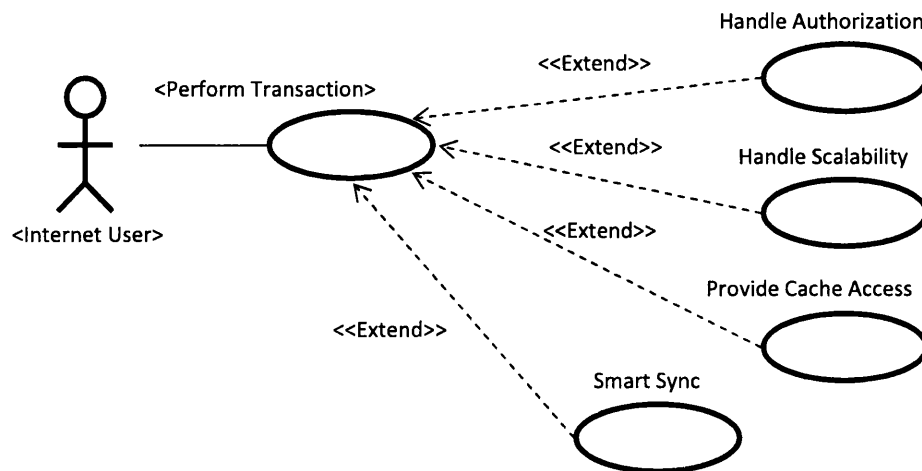


Figure 22 Structuring infrastructure use-cases.

4.4.3.7 Visualizing Infrastructure Use-Case Flows

Now that infrastructure use-cases have been identified, they can be described individually. Not all infrastructure use-case are as visible as Figure 23 and Figure 24. For example Figure 25 Handle Cache Access fulfils the NF2.1 performance – Application response requirement.

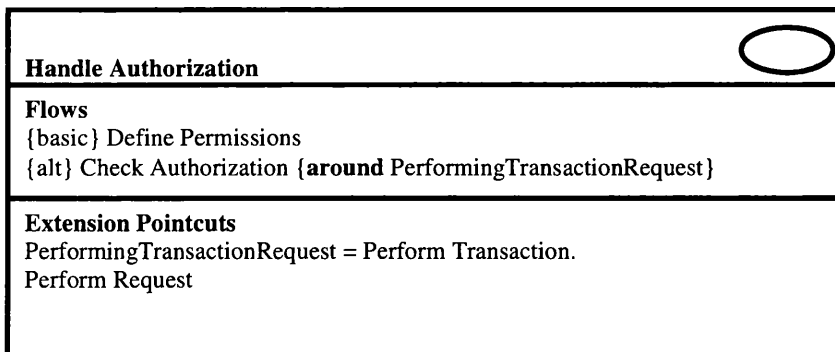


Figure 23 Handle Authorization use-case


Handle Scalability	
Flows {basic} Define type of support {alt} Check Availability of system { around PerformingScalabilityRequest}	
Extension Pointcuts PerformingScalabilityRequest = Perform Scalability. Perform Request	

Figure 24 Handle Scalability use-case


Handle Cache Access	
Flows {alt} Look Up Cache { around Around accessing data}	
Extension Pointcuts AccessingData = Perform Transaction. Access Frequently Used Data	

Figure 25 Provide Cached Access use-case


Smart Sync	
Flows {alt} Synchronize current environment {after PerformingRegistrationTransaction} {alt} Synchronize new environment {after PerformingRegistrationTransaction}	
Extension Pointcuts Synchronize current environment = Perform Registration. Perform Transaction Synchronize new environment = Perform Registration. Perform Transaction	

Figure 26 Smart Sync use-case

4.4.3.8 Analysis Model

The purpose of the analysis model is two-fold. Firstly, it is a refinement of the use-case model and secondly, it is where the description of the internal structure of the system begins. This assists to separate the infrastructure from the application. A separation that has to begin with the requirements and be preserved through analysis, design and implementation.

The language of the analysis model is a subset of the UML [144]. The analysis model provides three stereotyped analysis constructs: boundary, control and entity. A boundary construct is used to model the interaction between the system and the actors (i.e. users and external systems). Boundary constructs act as mediators between the system surroundings, it effectively shields the system from changes in its environment. If such changes occur, only boundary classes are affected. Control constructs are responsible for the coordination, sequencing, transaction, and control of other objects and is often used to encapsulate control related to a specific use-case. An instance of a control class often shares the lifetime as a use-case instance. Control constructs can also represent complex calculations and business logic.

An entity construct is used to model information in the problem domain. Such information is long-lived and often persistent. It encapsulates changes in the data structure. These analysis stereotypes shown in Figure 27 are used widely in the software development community [138, pp. 148-151] .

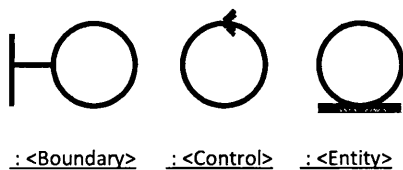


Figure 27 Analysis stereotypes

The User Account Activation scenario will be used as an example to demonstrate the separation of application and infrastructure concerns and their relationship in the design

phase. The application use-case will be analyzed alongside with couple of infrastructure use-cases. The participating analysis classes for the Account Activation use-case are

- Internet User as a : <Boundary> construct
- Application as a : <Boundary> construct
- Provisioning System as an : <Entity> construct
- Registration Presentation Function as a : <Control> construct
- Access Control Layer as an : <Entity> construct
- Access Presentation Function as a : <Control> construct

An Interaction diagram is selected instead of a Communication diagram if the use cases need be analysed in a greater detail. An interaction diagram shows how instances interact with each other in a chronological sequence from top to bottom and assist in identifying roles and responsibilities for class diagrams [138, pp. 192-194] . Figure 28 describes the chronological sequence that is important to the User Account Activation use-case. Step 3 of the User Account Activation use-case occurs around the {**around PerformingTransactionRequest**} pointcut identified earlier in the Handle Authorization infrastructure use-case. The Access Presentation Function checks if the details provide by the Internet User are correct and if they have sufficient authorization for the request to be performed. If any of them are incorrect an error message is presented. The Handle Authorization interaction is shown in Figure 29.

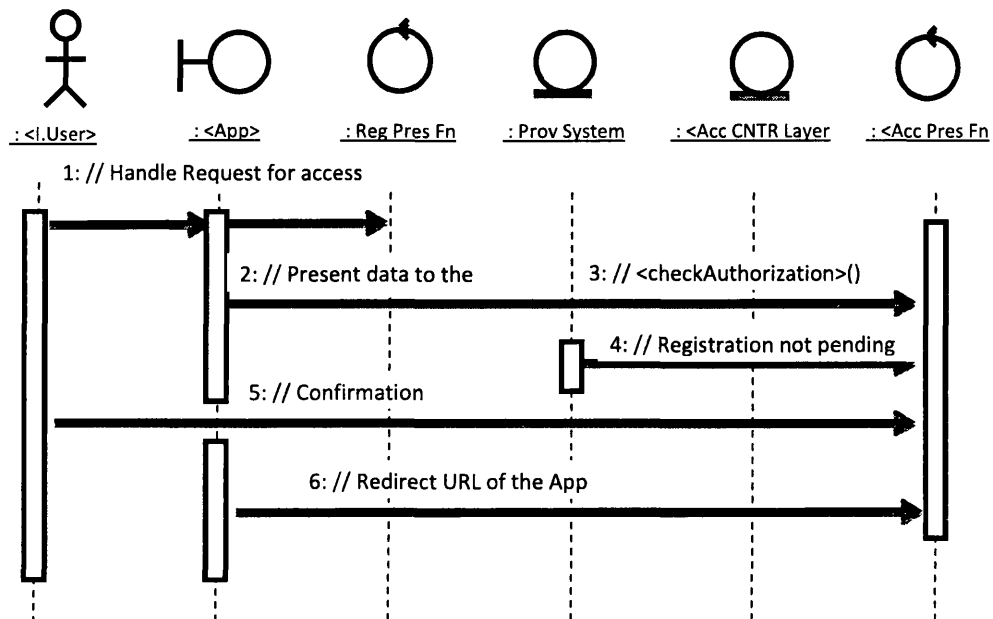


Figure 28 Interaction diagram for Account Activation use-case

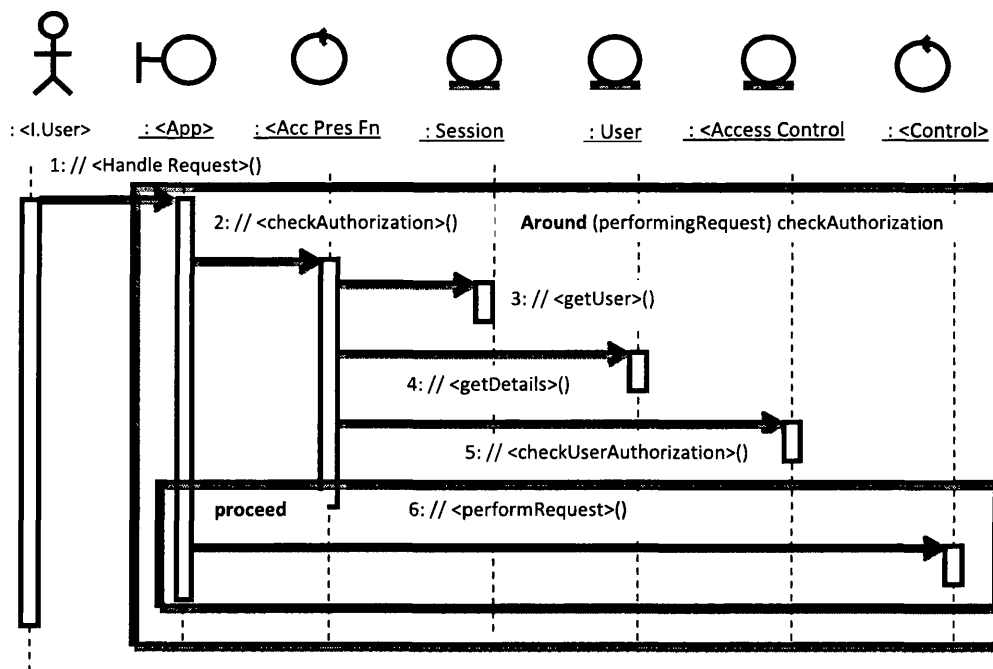


Figure 29 Interaction diagram for Handle Authorization

In a similar manner the interaction of other infrastructure use-case flow can be shown even if they are still at a conceptual stage. For example Figure 30 depicts the Interaction diagram for Smart Sync infrastructure use-case.

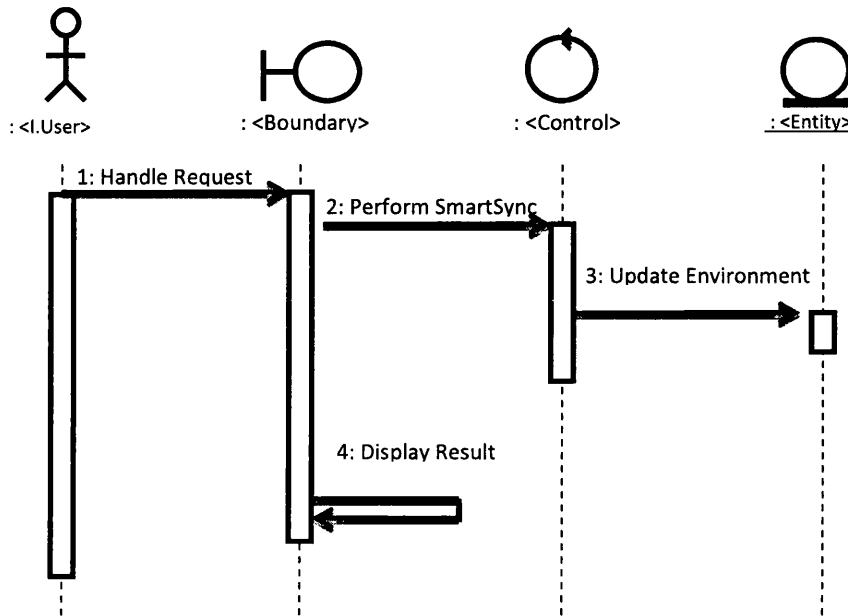


Figure 30 Interaction diagram for Smart Sync

4.4.3.9 Keeping Infrastructure Use-Cases Separate

Continuing with the Handle Authorization use-case, the model structure need to become further refined in to order to keep the concerns about authorization separate from the application use-case it extends. This can be achieved by putting all the related classes in a service packages. The Access Presentation Function and the Access control are for the sole purpose of handling authorization so they can be placed together in the infrastructure layer. The Session and User can be reused by other infrastructure services therefore they can be placed together in an infrastructure support package [138, pp. 246-249].

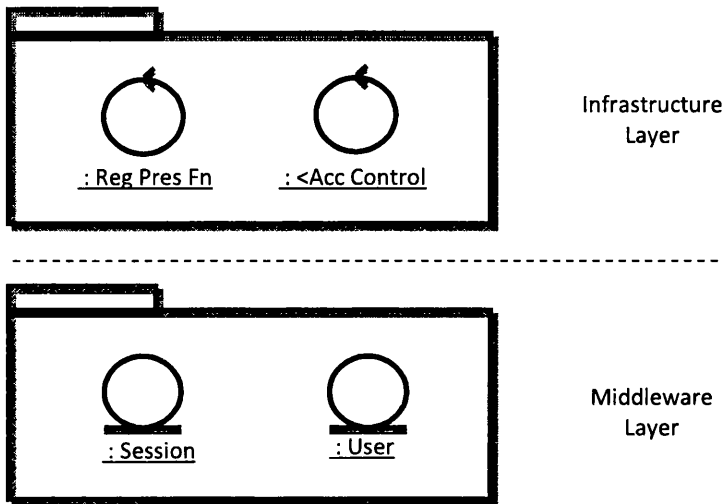


Figure 31 Infrastructure package for Handle Authorization use-case

The infrastructure can be kept separate by using a use-case slice that can comprise the classes and features that are specific to the realization of the Handle Authorization use-case and describe the interaction between the participating classes as shown in Figure 32. The Handle Authorization use-case slice also contains an extension of the boundary <App> class that it extends. This class extension is housed within an abstract aspect, HandleAuthorization. It is abstract because the pointcuts, though identified, are not defined. Therefore the HandleAuthorization aspect has to be specialised and attached to an actual use-case slice. This can be achieved through J2EE or AOP during implementation. It is beyond the scope of this case study to demonstrate how this can be achieved [138, pp. 252-256].

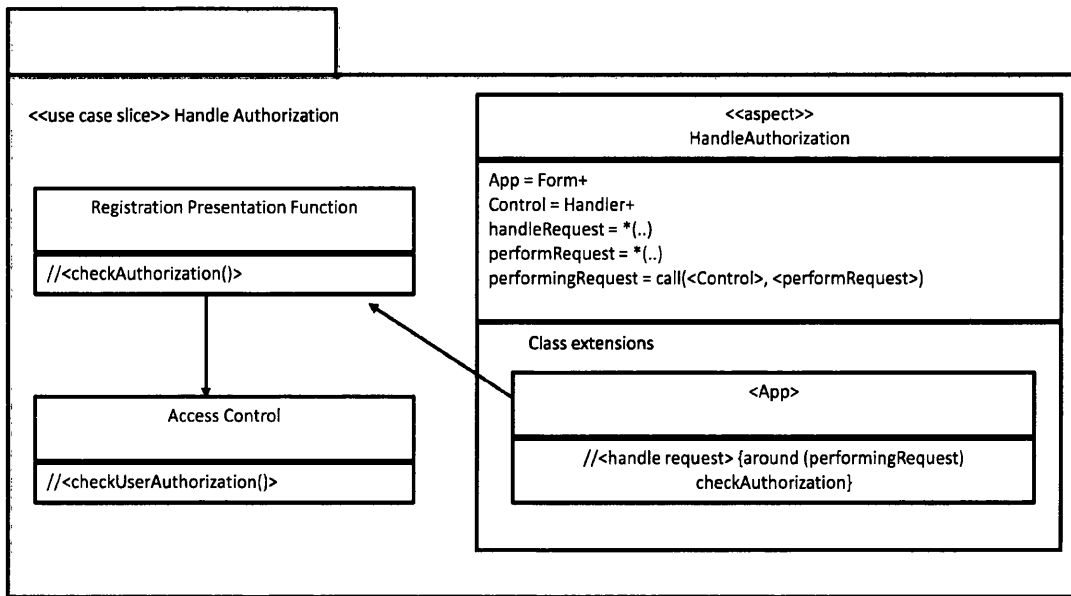


Figure 32 Use-Case slice Authorization [138, p. 249]

4.4.3.10 Conclusion

The study is based on a non-trivial new user provisioning system application adapted from an established company. The applications' purpose was to enable the company to reduce the cost and complexity of delivering changes. The study provides an outline of how to conduct AOSD with use-cases. Using the use-case driven approach allows the architects to explore the various ways in which a system is used, validating the stakeholders concern early in the project and drive the definition of the system architecture. This work is hoped to compliment the work already done by [140], [139] and [138]

The study starts discussing the functional and non-functional requirements gathering process, their relationship with application and infrastructure concerns, how to address stakeholders concerns and looking at the "typical" use-case models that are currently used by the industry. It then looks at use-case business scenarios that the architecture must achieve while maintaining all the requested requirements.

The suggested solution is a new way of visualizing and capturing application and infrastructure use case flows while keeping infrastructure separate from the application and infrastructure services separate from each other. This method results in assisting to build and evolve a system incrementally to meet the evolving needs of the stakeholders. This satisfies the decomposability criterion where a software construction helps in the task of decomposing a software problem into a small number of less complex sub-problems, connected by a simple structure, and independent enough to allow further work to proceed separately on each of them [25, p. 40]. Furthermore, it provides a better modular understandability thus assisting the maintenance of the implementation.

The use-case models that were analyzed also helped to verify that a resilient architecture is achieved by treating infrastructure use-cases as extensions of application use-cases. Use-case modelling was very useful in terms of having a high-level view of how these use-cases can be structured. Central to this approach was the use of the <Perform Transaction> use case pattern as a reference for analyzing infrastructure use cases. The result was a generic infrastructure use-case slice, a new modularity module, which can be specialized to attach to actual application use-case slices.

4.5 Summary

This chapter begins discussing the results of the research of two papers that the criteria that must be met in order to assess AOP as a software technique. In addition, three different case studies were selected to analyse real world none trivial applications discussing the benefits and drawbacks of the AOP technique. The first case study [68] provides a comparative analysis of the changes required to evolve the tangled and scattered versus aspect-oriented implementations. The second case study [80] presents an AOP implementation of a classical example of crosscutting concern known as persistence. The third case study outlines how to conduct AOSD with use-case driven approach. The suggested solution is a new way of visualizing and capturing application and infrastructure use case flows while keeping infrastructure separate from the application and infrastructure services separate from each other. A brief summary of the results are depicted below.

AOP was relatively a new concept in the time that [81] this research was conducted. However, there is an interesting discussion regarding the experience gained when evaluating a new software development technique. Murphy et al [81] starts its discussion by questioning a new software development technique in terms of its usability and usefulness. Validity, realism and cost were found to be the typical factors that are required when evaluating a method. In order for this to be quantified, various evaluation methods were introduced based on cost analysis highlighting some strengths and weaknesses of the various approaches. The importance of data gathering and analysis methods was briefly explained, particularly on experimental studies that are also applicable to any new programming technique in their early stage of development. This study can help determine if the technique is promising, and whether it can help direct the evolution of a technology to increase its usability and potential for usefulness.

Baniassad et al [96] was presented in the 1st international annual conference of AOSD in Enschede, the Netherlands [136] which as mentioned is the premier forum for the dissemination and discussion of AOSD ideas for both practitioners and researchers. The study was conducted to examine where developers encounter crosscutting code during a

program change task, and how the developers chose to manage that code. It was found that crosscutting code became an obstacle that developers had to manage when making the desired change.

When obstacle code related to a broader concern was encountered, developers had to try to understand both how the changes they were making affected the crosscutting concern, and how the crosscutting concern affected their change. Three strategies were used to deal with the crosscutting concern each corresponded to a different form of the obstacle code:

1. Change strategy - developers altered the crosscutting code to accommodate the change: This was used when there were suitable structural links and a developer could reason out from the obstacle point in the code related to the change to the concern code.
2. Within strategy - developers made the change work in the context of the crosscutting code: This was used when there were behavioural patterns but no structural links, developers reasoned from the concern code into the change code.
3. Working around - developers worked around the crosscutting code: When neither of these reasoning approaches was possible because of dense and implied code.

This paper also provides empirical evidence to support the existence and type of crosscutting concerns on which AOP approaches are based and set the basis for further examining of AOP

Coady et al., research [68] was presented in the 2nd international annual conference of AOSD in Boston, Massachusetts [136] states that changes to crosscutting concerns in an operating system are difficult to track. The study compares the evolution of four scattered and tangled concerns in kernel code with an aspect-oriented implementation of the same concerns. Localized changeability, explicit configurability, reduced redundancy and subsequent modular extensibility, are shown to be the key benefits of the aspect-oriented implementation assuming that they have negligible impact on performance.

A.Rashid et al., research [80] was also presented in the 2nd international annual conference of AOSD in Boston, Massachusetts [136] and is regarding persistence, a classical example of crosscutting concern. Persistence, the storing and retrieval of application data from non-volatile storage such as a file system or a relational database hasn't real world examples showing whether it can become an aspect and, if so, if it can be done in a way that is re-usable but ignored during application development. The paper uses a classical database application to show that persistence can be a highly re-usable aspect and be developed into a general aspect-based persistence framework. Furthermore, persistence has to be considered when designing the architecture of data-consumer components where such components need to account the declarative nature of retrieval mechanisms used by many database systems and deletion operation during application design because is highly triggered by most applications.

The use-case based driven approach with AOSD study is based on the candidates' experience and research. The case study was based on a non-trivial new user provisioning system application adapted from an established company. The study provides an outline of how to conduct AOSD with use-cases. It shows that it is possible to identify trade-offs among broadly scoped properties early on in the development cycle and therefore providing decision support for the stakeholders involved. At the same time, being based on use-cases, the approach adheres to the industry standard hence making it suitable for incorporation in existing requirements engineering practices.

The suggested solution is a new way of visualizing and capturing application and infrastructure use case flows while keeping infrastructure separate from the application and infrastructure services separate from each other. This results in assisting to build and evolve a system incrementally to meet the evolving needs of the stakeholders. The use-case models that were analyzed also helped to verify that a resilient architecture is achieved by treating infrastructure use-cases as extensions of application use-cases. Central to this approach was the use of the <Perform Transaction> use case pattern as a reference for analyzing infrastructure use cases.

5. Conclusion and Future Work

In this chapter, points discussed throughout this thesis and final thoughts are brought together with an overview of potential avenues for further work.

5.1 Conclusion and Discussion

The thesis started with introducing the concepts of modularization as an instrument for improving the flexibility, efficiency, extendibility, reusability and comprehensibility of a system while allowing the shortening of its development time. It was showed that modular programming can be achieved when criteria such as decomposability, composability, understandability, continuity and protection are met, while the sustainability of a modularity requires direct mapping, fewer, smaller and explicit interfaces and information hiding. Assumptions about software design processes and programming languages were discussed and it was shown that a design process and a programming language work well together when the programming language provides abstraction and composition. These methods can cleanly support the kinds of units the design process breaks the system into and a clear and simple one-to-one mapping from design level concepts to their source code implementation. AOP was suggested because it offers a clear and simple one-to-one mapping from design level concepts to their source code implementation which also helps the program to be simpler to understand and maintain.

However, some papers argued [33, pp. 1-4] that AOP languages do not provide the third point of the benefits quoted by Parnas [9] i.e. comprehensibility, because they require systems to be studied in their entirety. Also in [34, p. 327] arguing for AOP, states that the modularity of a system should reflect the way developers would like to think about modularity, rather than the way in which developers are forced to think about it due to the language or other tools. Current aspect-oriented languages such AspectJ, however, do have tools and mechanisms that compensate this lack of modularity. Furthermore, a

preliminary evaluation has showed [33, p. 11] that with some modifications the language can provide sufficient flexibility according to second criteria of Parnas. Lopes also found in [137] that under the theory of modularity [9], certain aspect-oriented modularizations can add value to the design.

The research attempted to explore and analyse the state-of-the-art in AOP techniques that would provide the tools to assess and compare AOP versus other programming approaches. As first step to achieve this goal was to survey AOP technologies, frameworks and investigate language models and meta-models for AOP. This would allow a more general but comprehensive comparison and analysis of the fundamental aspect language features as well as their implementation and execution techniques. However, the AOSD Languages Lab had already performed an extensive survey on twenty seven AOP languages according to particular dimensions of interest ensuring that each language is appropriately reviewed and the commonalities and the variations of each language identified. This was very important as this survey can be used as an input on the classification of aspect languages and a common metamodel. The survey consisted of two different categories the first was the language model where the focus is the language itself and the latter was the execution model where the focus is on the implementation of the woven code.

Next, each language and execution model in the survey had to be described among the same dimensions of interest. These dimensions were essentially derived from filtering the major commonalities and variations between the surveyed aspect languages. The view of this dimensions are the join point model, pointcut language, advice model and language, aspect module and composition model and aspect instantiation model. These dimensions were the building blocks of a common metamodel for AOP languages as an open and extensible framework that will allow collaboration and integration activities between the designers of these languages and categorize aspect languages according to the common language concepts and their semantics. The common language concepts framework metamodel (common metamodel) consisted of four sub-metamodels, namely the join point, pointcut, aspect binding and advice metamodels. The metamodel took a

framework approach in order to avoid oversimplification as specific language features of particular aspect languages can only be partially described as specializations of the concepts described in the common metamodel which was essential because it allows the users to describe specific features of aspect languages as specializations of the framework.

An interesting observation that was found while looking at the results of the languages lab [54] was that most of the aspect languages that exist today have an object-oriented language as their base language, therefore, a particular focus was made because particular properties are exhibited at join points and these properties depend on the paradigm of the base language and the kind of join point that is used. Therefore, the general concept of a join point is effectively covered in the metamodel as a point in the execution of a program but needs to specialise this general notion in order to reflect the different kinds of join points available in different aspect languages.

It was then shown that the description of the semantics of the metamodel can be done by using the implementation of an interpreter because the set of evaluation functions defined by the interpreter can have a close relation with its description using operational semantics and the interpreter can provide executable semantics which establishes a solid ground for tools to investigate and experiment with the semantics of language features. The concepts that interpreter employs to explain the semantics of the metamodel are the base and metalevel aspect interpreter, discrete evaluation through join point stepping, continuations, woven execution of applications, metalevel operations, metalevel aspect state, and aspect environment.

The research showed that because aspects impose a different behaviour on the base program, an integrated behaviour of the base and aspect programs is required. This can be achieved when the metalevel aspect interpreter that interprets the aspect-oriented part of the program in a metamodel representation, controls the execution of the base interpreter which, interprets the base program part. As a result, the execution of the aspect program essentially modifies the execution of the base program [54, p. 15].

Furthermore, the notion of continuation was the most essential concept to model the execution semantics of aspect languages because it captures the current execution state of the program such that it can be stored and reconstructed later on. Also the advice metamodel and the metalevel operations are embedded in the advice and these metalevel operations are executed but not understood by the base interpreter. Therefore, the base interpreter's execution must be halted in order to execute the metalevel operations by the aspect interpreter.

In terms of the classification of aspect languages, some improvements are required in the initial mapping of different language features into the metamodel. For example the syntax and structure of a language have not been taken into account in the metamodel. Although the initial metamodel was not intend to do that, structure and syntax have a significant impact on the expressiveness and identification of a language. Furthermore, the Aspect Sandbox (ASB) [71] has similar approach with this work apart that the way that the interpreter execution semantics is considered without any weaving. On the contrary, the explicit setup of the metamodel and its interpreter is a complete interpreted execution.

Next, the aim of the research when analysing the non-trivial applications was to introduce a set of evaluation techniques that would enable the assessment of any new software methodology, while trying to understand the usability and usefulness, the strengths and weaknesses of these methods and the current strategies that are in place in order deal with crosscutting concerns.

Murphy et al. [81] research was chosen as a first study not only because of its historical value as it was the first assessment of its kind in terms of AOP, but it sets the criteria that one needs to have prior starting any assessment of a new software technique and introduced some important discussion regarding empirical research methods. Although the research was effective in terms of assessing whether and how AOP might ease some development tasks it is important to note that AOP is not trying to replace OOP but to capture important design decisions that are difficult to capture in the traditional OOP

environment. (i.e. a new programming technique [36]. Therefore, the experiments although exploratory, would yield better results if focused on issues such as crosscutting concerns.

Perhaps more interesting results could have been taken by a similar case study known as ATLAS [138] that was conducted at the same era that [81] took place. The application was fairly moderate and was built initially in C++ and then in AOP using AspectJ. The results were positive in favor of AOP but some lessons were learned. In brief the lessons learnt were that it was found easier to manage the evolution of the system when classes were not coupled to aspects. Class directional aspects facilitated the readability, modifiability, and reusability of class and aspect code something also mentioned in chapter 2. It made it easier to reason about and test when the aspect code is kept simple, clear and with a well-defined scope. Using dynamic aspects provide runtime configurability, but can complicate system set-up code. It is important to maintain a stand-alone object model, which aspects extend and finally, the most important lesson was that the hardest decision facing a developer working with AOP is determining what should be an aspect and what should be a class. In the beginning of the ATLAS development, it was thought that the implementation would have many more aspects but in most of the cases, while implementing an aspect it was found that with some straightforward changes to the object model could accomplish the same goal more effectively.

In the case of research of Baniassad et al. [96] the results showed that when performing a task at certain points the developer needed to see the behavioural effects of aspects on methods of interest. Similar results were found also on a case study of AspectJ by [139]. Also developers found it difficult to reason about a separated concern when the interface between the core code and the concern code was too broad i.e. the more constrained and defined the interface, the easier it was for developer to determine the area of influence between the code and concern code. This result was also verified by [90].

The first case study [68] provided a comparative analysis of the changes required to evolve the tangled and scattered versus aspect-oriented implementations and had

positive results. It confirmed that AOP could improve the evolvability of OS code but there were some issues that limit the validity of this research. In summary, the focus was only on the evolution of specific concerns in isolation rather than producing full successive versions of the OS code. The concerns were evolved by a single developer for all the versions. An in depth cost/benefit analysis was still required because improving modularity of operating systems will not be meaningful if aspects substantially reduce performance. Finally it imperative to determine the precise costs associated with more sophisticated compositions of aspects relative to their current implementation.

This research decided to classify AspectC according to the metamodel as this would help resolving this limitation because of a better understanding of the aspect language features, strengths and weaknesses. It would also assist in the creation of a tool to assist the indirect method for the textual locality in terms of changeability.

The second case study [80] presents an AOP implementation of a classical example of crosscutting concern known as persistence. The aim of this study was to assess if AOP techniques offer an effective means to modularise persistence in a real world application scenario. The outcome was positive with a number of important software engineering factors to keep in mind. First, the necessity of the trade-offs between generalization and performance. The application used reflection which allowed for generalization and reusability of the SQL translation mechanism i.e. the aspectised persistence mechanism. Well modelled aspects require investigation the suitability of the available techniques for implementing the various concerns within the aspect. For example, the use of AspectJ constructs to identify points where persistence-related behaviour has to be composed, while reflection has been used to keep the SQL translation generic and avoid duplication of transaction code during database access.

However, the choice of suitable technique is limiting the available tools and the way they interact. So instead of using composition filters, AspectJ introductions were used. Some of these results verify the results shown in the Atlas case study. The research also

found that a persistence aspect can be designed so it can be reusable. This can be done by utilizing the suggested persistence framework but the reuse of the framework should be strengthened by reuse of specification which clearly defines the interface of aspects behaviour. Finally, it was also showed that an application and a persistence aspect can be partially developed independently of each other. For example storage does not need to be considered but retrieval is essential. The most important factor is allowing a natural separation of concerns while developing the persistence infrastructure and keeping the reusability and application independence requirements.

Also the claims about advantages and disadvantages of aspect technologies are quite broad. The main problem of aspect technologies, whatever approach is considered, is not just about crosscutting or separation of concerns, but it involves deeper research about how to understand a number of software parts as separated objects and then integrate some of them into a coherent system. This situation also bears the issue of locality of changes, because the more interactions with other components (or aspects) the developer has to know in order to understand the system, the more complex the maintenance of this software results.

Finally, the third case study is a new contribution towards the AOSD community. The study provides an outline of how to conduct AOSD with use-cases allow the architects to explore the various ways in which a system is used, validating the stakeholders concern early in the project and drive the definition of the system architecture. This continues the work already done in this field [139] and [138]. However, using the Jacobson et al. [140] methodology this work is furthered by introducing a new way of visualizing and capturing application and infrastructure use case flows while keeping infrastructure separate from the application and infrastructure services separate from each other. This results in assisting to build and evolve a system incrementally to meet the evolving needs of the stakeholders.

5. 2 *Further Work*

There are few potential avenues for further research.

1. AOP is known to have a solution for concerns such as logging, tracing, transaction management, security, caching, error handling, performance monitoring, custom business rules [41]. This research started from the beginning of the developments of AOP and there are still important non-trivial applications to investigate. One of them is design patterns and pattern composition as it has been shown as a challenge to apply design patterns in real software systems. One of the main issues is that multiple design patterns in a system are not limited to affect only the application concerns. They also crosscut each other in multiple varied ways so that their separation and composition are not an easy task. In this perspective, it is of vital importance to systematically verify whether AOP supports improved composability of design patterns [140], [141], [142]. Another classic example is studying idioms-based implementations of crosscutting concerns in the context of a real-world, large-scale embedded software system analysing apparently simple concerns such as tracing [143].
2. Further classification according to the metamodel would help understanding better the aspect language features, strengths and weaknesses but also the experimental interpreter of the metamodel 'Metaspin' requires more development to render it into a complete experimental vehicle [69].
3. To approach the question of language integration from the formal viewpoint, and discuss the differences between the CASB model and the metamodel as shown in [64], [144]
4. To investigate other AOP approaches not in the scope of AOSD such as Spring Source framework.
5. Further work can be done on the use-case driven approach by implementing the use-cases introduced in the study.

Bibliography

- [1] Assembly. (1950). *History of Computer Languages and Their Evolution*. Retrieved 2009, from <http://www.scriptol.com/programming/history.php>
- [2] Fortran. (1953). *Manual Fortran* . Retrieved 2009, from <http://www.fh-jena.de/~kleine/history/languages/FortranAutomaticCodingSystemForTheIBM704.pdf>
- [3] J.W.Backus, J. H.Wegstein, A.vanWijngaarden, M.Woodger, F. L.Bauer, J.Green, C.Katz, J.McCarthy, A.J.Perlis, H.Rutishauser, K.Samelson, B.Vauquois. (1960). Report on the Algorithmic Language ALGOL 60. *Communications of the ACM* , 3 (5), 299-314.
- [4] K.Nygaard,O.J.Dahl. (1978). The development of the SIMULA languages. *ACM SIGPLAN Notices* , 13 (8), 245 - 272.
- [5] B.Liskov, A.Snyder, R.Atkinson, C.Schaffert. (1977). Abstraction mechanisms in CLU. *Communications of the ACM* , 20 (8), 564 - 576.
- [6] E.W.Dijkstra. (1976). *A Discipline of Programming*. Prentice Hall, Inc.
- [7] G.C.Murphy, R.J.Walker, E.L.A.Baniassad, M.P.Robillard, A.Lai, M.A.Kersten. (2001). Does aspect-oriented programming work? *Communications of the ACM* , 44 (10), 75 - 77.
- [8] B.W.Kernighan, D.M.Ritche. (1988). *The C Programming Language: Second Edition*. Englewood, New Jersey: Prentice Hall.
- [9] D.L.Parnas. (1972). On the Criteria To Be Used in Decomposing System into Modules. 15, pp. 1053-1058.
- [10] D.H.Ingalls. (1978). The Smalltalk-76 Programming System Design and Implementation. *5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, (pp. 9-16). Tucson.
- [11] G.Kiczales, J.d.Rivieres, D.G.Bobrow. (1991). *The Art of the Metaobject Protocol*.
- [12] G.Kiczales. (1996). Beyond the black box: open implementation. *Software, IEEE* , 13 (1), 8, 10-11.
- [13] H.Schildt. (1983). *C++ The Complete Reference* (3rd ed.). Osborne McGraw-Hill.
- [14] Ada Europe. (1983). *Ada Europe*. Retrieved from <http://www.ada-europe.org/>

- [15] M.Tsukamoto, Y.Hamazaki, T.Nishioka, H.Otokawa. (1996). *The version management architecture of an object-oriented distributed systems environment:OZ++*. Springer Berlin / Heidelberg.
- [16] S. Ruthfield. (1995). *The Internet's History and Development*. Retrieved 2008, from <http://www.acm.org/crossroads/xrds2-1/inet-history.html>
- [17] Java Sun. (1995). *Java*. Retrieved 2004, from <http://java.sun.com>
- [18] AOSD Europe. (2002). *European Network of Excellence on Aspect-Oriented Software Development*. Retrieved 2005, from <http://www.aosd-europe.net/>
- [19] Harrison, Ossher. (1994). Subject-oriented programming: Supporting Decentralized Development of Objects. *7th (IBM) Conference of Object Technology*.
- [20] H.Ossher, P.Tarr. (1999). Multi-dimensional separation of concerns and the hyperspace approach. *Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*.
- [21] G.Kiczales, J.Lamping, A.Mendhekar, C.Maeda, C.V. Lopes, J. Loingtier, J. Irwin. (1997). Aspect-Oriented Programming. *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)* (pp. 220-242). Springer.
- [22] R.Gauthier, S.Pont. (1970). *Designing Systems Programs*. Englewood Cliffs, N.J.: Prentice-Hall.
- [23] D.L.Parnas. (1971). *On the Criteria to be Used in decomposing Systems into Modules*. Technical, Department of Computer Science, Carnegie-Mellon U, Pittsburgh, Pa.
- [24] S.Schach . (2006). *Object-Oriented and Classical Software Engineering*. Seventh Edition. McGraw-Hill.
- [25] B.Meyer. (1997). *Object-oriented software construction* (2 ed.). Upper Saddle River, N.J: Prentice Hall.
- [26] N.Wirth, C.A.R.Hoare. (1966). A Contribution to the development of ALGOL. *Communications of ACM* , 9 (6), 413-431.
- [27] A.Colyer, A.Clement, G.Harley, M.Webster. (2005). *Eclipse AspectJ*. Addison-Wesley.

- [28] T.Elrad, R.E.Filman, A.Bader, M.Aksit, G.Kiczales, K.Lieberherr, H.Ossher. (2001). Aspect-oriented programming. *Communications of the ACM* , 10.
- [29] R.Miles. (2004). *AspectJ Cookbook, Real-World Aspect-Oriented Programming with Java*, (1st ed.). O'Reilly.
- [30] A.Rashid, A.Moreira, J.Araujo, P.Clements, E.Baniassad, B.Tekinerdogan. (2005). *Early-aspects*. Retrieved 2008, from <http://www.early-aspects.net/>
- [31] A. Rashid, A. Moreira, and J. Araujo. (2003). Modularisation and Composition of Aspectual Requirements. *Proceedings of 2nd International Conference on Aspect-Oriented Software Development (AOSD)* (pp. 11-20). ACM.
- [32] E. L. A. Baniassad and S. Clarke. (2004). An Approach for Aspect-Oriented Analysis and Design. *Proceedings of International Conference on Software Engineering (ICSE)* (pp. 158-167). IEEE Computer Society.
- [33] M.Mezini, K.J. Lieberherr. (1998). Adaptive Plug-and-Play Components for Evolutionary Software Development. *OOPSLA*, (pp. 97-116).
- [34] AOSD Europe. (2002). *European Network of Excellence on Aspect-Oriented Software Development*. Retrieved 2005, from <http://www.aosd-europe.net/>
- [35] C.Clifton, G.T.Leavens. (2002). *Leavens Observers and Assistants: A Proposal for Modular Aspect-Oriented Reasoning*. Department of Computer Science Iowa State University.
- [36] G.Kiczales, E.Hilsdale, J.Hugunin, M.A.Kersten, J.Palm, W.G.Griswold. (2001). An Overview of AspectJ. *ECOOP. LNCS 2072*, pp. 327-353. Springer-Verlag.
- [37] D.L.Parnas. (1972). *On the Criteria To Be Used in Decomposing System into Modules*. *Communications of the ACM*.
- [38] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J. Loingtier, J. Irwin. (1997). Aspect-Oriented Programming. *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)* (pp. 220-242). Springer.
- [39] T.Elrad, R.E.Filman, A.Bader. (2001). Aspect-oriented programming. *Communications of the ACM* .
- [40] J.Zhicheng. (2005). *Java news brief* . Retrieved 2007, from Ociweb - Object Computing, Inc.: <http://www.ociweb.com/jnb/jnbNov2005.html>

- [41] AspectJ. (2002). *AspectJ*. Retrieved 2004, from <http://www.eclipse.org/aspectj/>
- [42] JBoss. (2001). *JBoss*. Retrieved from <http://www.jboss.org/>
- [43] SpringSource. (2002). *Spring Source*. Retrieved 2006, from <http://www.springsource.org/>
- [44] Nanning. (2002). *Nanning*. Retrieved 2006, from <http://nanning.codehaus.org/overview.html>
- [45] Java Dynamic Proxy Classes. (2004). *Java Dynamic Proxy Classes*. Retrieved 2006, from <http://java.sun.com/j2se/1.5.0/docs/guide/reflection/proxy.html>
- [46] Sourceforge. (2004). *Code Generation Library*. Retrieved 2007, from <http://cglib.sourceforge.net/>
- [47] AOSD Conference. (2002). *1st International Conference on Aspect-Oriented Software Development*. Retrieved 2005, from Program: <http://trese.cs.utwente.nl/aosd2002/index.php?content=program>
- [48] TAOSAD. (2002). *The Aspect-Oriented Software Architecture Design Portal*. Retrieved 2005, from <http://trese.cs.utwente.nl/taosad/aosd.htm>
- [49] HyperJ. (1999). *HyperJ*. Retrieved 2004, from <http://www.alphaworks.ibm.com/tech/hyperj>
- [50] ComposeJ. (1999). *ComposeJ*. Retrieved 2007, from ComposeJ: <http://trese.cs.utwente.nl/oldhtml/publications/paperinfo/wichman.thesis.pi.top.htm>
- [51] DemeterJ. (2002). *DemeterJ*. Retrieved 2008, from <http://www.ccs.neu.edu/research/demeter/sources/DemeterJava/product-guide.html>
- [52] D12 – Language Lab. (2005). *Survey of Aspect-oriented Languages and Execution Models*. (M. J. Brichau, Ed.) AOSD-Europe.
- [53] J. Brichau, M. Haupt. (2005). *A Taxonomy of Aspect Language Features*. Technical report of AOSD-Europe.
- [54] Sun Services. (2002). *SJCP Course SL-275*. Sun Microsystems Inc.
- [55] AOSD Europe, Atelier Research. (2002). *Atelier Research*. Retrieved from <http://www.aosd-europe.net/> Industry » Collaborative Project: http://gateway.comp.lancs.ac.uk:8080/c/portal/layout?p_l_id=1.57

- [56] D39 – Language Lab. (2006). *An Initial Metamodel for Aspect-Oriented Programming Languages*. (M. (. J.Brichau (INRIA/VUB), Ed.) AOSD-Europe.
- [57] J.Fabry, D.Rebernak, T.Cleenewerck, A.FLemEUR, J.Noy'e, E.Tanter. (2007). Summary of the Second Workshop on Domain-Specific Aspect Languages. *ACM* .
- [58] B.Harbulo, J.R.Gurd. (2006). A join point for loops in AspectJ. *AOSD 2006 Conference*.
- [59] P.Bekaert, G.Delanote, F.Devos, E.Steegmans. (2002). *Specialization/Generalization in Object-Oriented Analysis: Strengthening and Multiple Partitioning* . Springer Berlin / Heidelberg.
- [60] T.Skotiniotis, K.Lieberherr, D.H.Lorenz. (2003). Aspect Instances and their Interactions. *AOSD 2003 Workshop on Software-engineering Properties of Languages for Aspect Technologies*. Boston.
- [61] E.Ernst, D.H.Lorenz. (2003). Aspects and polymorphism in AspectJ. *AOSD 2003 Conference*. Boston.
- [62] A.Aaby. (2004). *Introduction to Programming Languages*. Retrieved 2009, from <http://web.archive.org/web/20040410154109/cs.wwc.edu/~aabyan/PLBook/HTML/Semantics.html>
- [63] G.Winskel. (1993). *The Formal Semantics of Programming Languages: An Introduction* . MIT Press.
- [64] D.B.Tucker, S.Krishnamurthi. (2003). Pointcuts and advice in higher-order languages . *AOSD Conference 2003*, (pp. 158–167).
- [65] J.C.Reynolds. (1993). The Discoveries of Continuations. *LISP AND SYMBOLIC COMPUTATION* , 6, 233{247.
- [66] D41 - AOSD Europe. (2006). *Common Aspect Semantics Base*. S.D.Djoko, R.Douence, P.Fradet, D.L.Botlan.
- [67] AOSD Europe, CASB. (2002). *AOSD Europe,CASB*. Retrieved 2008, from <http://www.aosd-europe.net/> » Research » Formal Methods » CASB: http://gateway.comp.lancs.ac.uk:8080/c/portal/layout?p_l_id=1.26
- [68] AspectC. (2002). *AspectC - The software Practice lab*. Retrieved 2008, from <http://www.cs.ubc.ca/labs/spl/projects/aspectc.html>

- [69] AspectC++. (2001). *The Home of AspectC++*. Retrieved 2009, from AspectC++:
<http://www.aspectc.org/>
- [70] Y.Coady, G.Kiczales. (2003). Back to the future: a retroactive study of aspect evolution in operating system code. *International Conference on Aspect-Oriented Software Development (AOSD)* (pp. pp. 50-59). ACM New York, NY, USA.
- [71] D55 – Language Lab. (2006). *Classification of Surveyed Aspect Languages according to the Aspect Languages Metamodel*. (T. (. J.Brighau (INRIA/VUB), Ed.) AOSD-Europe.
- [72] M.Y.Coady. (2003). *Improving evolvability of operating systems with aspectC*. Thesis - University of British Columbia.
- [73] C.Dutchyn, G.Kiczales, H.Masuhara. (2002). *Aspect Sandbox*.
- [74] AOSD, Sandbox. (2002). AOP Language Exploration Using the Aspect Sand Box. *AOSD 2002 Conference*. Tutorial.
- [75] A.Denmark . (2005). *Ambient Computing in a Critical, Quality of Life Perspective*. Retrieved 2009, from <http://www.daimi.au.dk/~olavb/AQLWS/>
- [76] C.V.Lopes, G.Kiczales. (1998). Recent Developments in AspectJ. *European Conference on Object-Oriented Programming (ECOOP)*, 1543, pp. 398-401.
- [77] C.V.Lopes. (1997). *A Language Framework for Distributed Computing*. Boston: College of Computer Science, Northeastern University.
- [78] C.V. Lopes, K.J.Lieberherr. (1994). Abstracting process-tofunction relations in concurrent object-oriented applications. *European Conference on Object-Oriented Programming (ECOOP)*, 821, pp. 81-99.
- [79] M.Askit, L.Bergmans, S.Vural. (1992). An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach. *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, Vol. 615, pp. 372-395.
- [80] H.Ossher, M. Kaplan, A.Katz, W.Harrison, V. Krnskal. (1996). *Specifying subject-oriented composition* (Vols. Vol. 2, No. 3.). TAPoS.
- [81] P. Tan, H.Ossher, W.Harrison, S.M. Sutton. (1999). N degrees of separation: Multi-dimensional separation of concerns. *Preceedings of the 21 st International Conference on Software Engineering*, (pp. pp. 107-119).

- [82] G.C.Murphy, R.J.Walker, E.L.A.Baniassad. (1999). Evaluating Emerging Software Development Technologies:Lessons Learned from Assessing Aspect-oriented Programming. *IEEE Transactions on Software Engineering* Vol. 25, No. 4, pp. 438-455.
- [83] S.L.Peeger. (1994). *Design and analysis in software engineering, part 1: The language of case studies and formal experiments*. ACM SIGSOFT Software Engineering Notes, 19(4).
- [84] M.V.Zelkowitz, D.R.Wallace. (1998). *Experimental models for validating technology*. Computer, 31(5).
- [85] R.K.Yin. (1994). *Case Study Research: Design and Methods*. Thousand Oaks, CA: Sage Publications.
- [86] B.Curtis, S.B.Sheppard, E.Kruesi-Bailey, J.Bailey, D.A.Boehm-Davis. (1989). Experimental evaluation of software documentation formats. *Journal of Systems and Software* , 9 (2).
- [87] A.A.Porter, H.P.Siy, C.A.Toman, L.G.Votta. (1997). An experiment to assess the cost benefits of code inspections in large scale software development. *IEEE Transactions onSoftware Engineering* , 23 (6).
- [88] M.A.D.Storey, K.Wong, P.Fong, D.Hooper, K.Hopkins, H.A. Muller. (1996). On designing an experiment to evaluate a reverse engineering tool. *Proceedings of the Third Working Conference on Reverse Engineering*. IEEE Computer Society, Press.
- [89] B.Schneiderman. (1989). *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Reading, MA: Addison-Wesley Publishing Co.
- [90] V.R.Basili, R. D. (1986). *Experimentation in software engineering*. IEEE Transactions on Software Engineering.
- [91] R.J.Walker, E.L.A.Baniassad, G.C.Murphy. (1999). An Initial Assessment of Aspect-oriented Programming. *Proceedings of the 21st International Conference on Software Engineering*.
- [92] G.C.Murphy, E.L.A.Baniassad . (1997). *Qualitative case study results*. UBC-CS-SE-AOP.
- [93] J.E.McGrath. (1995). *Methodology matters: Doing research in the behavioral and social sciences* (2nd ed.). (J. G. In R.M. Baecker, Ed.) Morgan Kaufmann Publishers,Inc.

- [94] R.J.Walker, E.L.A.Baniassad,G.C.Murphy. (1998). *Assessing AOP & design: Preliminary results*. University of British Columbia.
- [95] A.V.Mayrhauser, A.M.Mans. (1996). *Identification of dynamic comprehension processes during large scale maintenance* (Vol. 22(6)). IEEE Transactions on Software Engineering.
- [96] M.A.Kersten, G.C.Murphy. (1999). Atlas: A Case Study in Building a Web-based Learning Environment using Aspect-oriented Programming. *Object-oriented Programming Systems, Languages and Applications Conference (OOPSLA)*. 34(10), pp. 340-352. ACM, SIGPLAN Notices.
- [97] E.L.A.Baniassad, G.C.Murphy, C.Schwanninger M. Kircher. (2002). Managing Crosscutting Concerns during Software Evolution Tasks: An Inquisitive Study. *1st International Conference on Aspect-Oriented Software Development(AOSD)* (pp. 120-126). ACM.
- [98] T.Lethbridge, S.Sire, J.Singer. (2000). *Studying Software Engineers:Data Collection Methods for Software Field Studies*. Empirical Software Engineering.
- [99] BW.Kernighan, D.M.Ritche. (1988). *The C Programming Language: Second Edition*. Englewood, New Jersey: Prentice Hall.
- [100] B.Stroustrup. (1991). *The C++ Programming Language: Second Edition*. AddisonWesley Publishing Co.
- [101] K.Arnold, J.Gosling. (1996). *The Java Programming Language*. ACM Press Books, Addison Wesley Longman.
- [102] R.Brookes. (1983). Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine studies* (18), 543-554.
- [103] E.Soloway, K. Erlich. (1989). Empirical studies of programming knowledge. *SE-10* (5), 595-609.
- [104] B.Schneiderman, R.Mayer. (1979). Syntactic/semantic interactions in programmer behaviour: A model and experimental results. *International Journal of Computer and Information Sciences* , 8 (3), 219-238.
- [105] H.Permington. (1987). Stimulus structures and mental representations in expert comprehension of computer programs . *Cognitive Psychology* , 19, 295-341.

- [106] S.Letovsky. (1986). Cognitive Processes in Program Comprehension In Empirical Studies of Programmers.
- [107] A.van Mayrhanser, A.Vans. (1994). Comprehension processes during large scale maintenance. *In Proceedings of the 16th International Conference on Software Engineering*, (pp. 39-48).
- [108] R.J.Walker, E.L.A.Baniassad, G.C.Murphy. (1999). An Initial Assessment of Aspen-Oriented Programming. *21st International Conference on Software Engineering*, (pp. 120-130).
- [109] L.L. Lehman, L.A. Belady. (1985). *Program Evolution*. APIC Studies in Data Processing, Volume 3.
- [110] FreeBSD. (2009). *FreeBSD*. Retrieved 2009, from <http://www.freebsd.org/doc>
- [111] J. Belzer, A.G.Holzman,A.Kent. (1981). *Virtual memory systems*. Encyclopedia of computer science and technology, 14, CRC Press.
- [112] IBM. (2009). *IBM*. Retrieved 2009, from <http://publib.boulder.ibm.com/infocenter/db2luw/v9r5/index.jsp?topic=/com.ibm.db2.luw.admin.perf.doc/doc/c0009651.html>
- [113] Y.Coady, G.Kiczales, M.Feeley,G.Smolyn. (2001). Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code. *Proceedings of the Joint European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-9)* .
- [114] IBM. (2009). *Publib boulder IBM*. Retrieved 2009, from http://publib.boulder.ibm.com/infocenter/pseries/v5r3/index.jsp?topic=/com.ibm.aix.security/doc/security/disk_quota.htm
- [115] J.Blazewicz, K.H.Ecker, E.Pesch, G.Schmidt,J.Weglarz . (2001). *Scheduling Computer and Manufacturing Processes*. Berlin : Springer.
- [116] NetBSD. (2007). *NetBSD Kernel Developer's Manual*. Retrieved 2009, from <http://www.daemon-systems.org/man/tsleep.9.html>
- [117] A.Chou, J.Yang, B.Chelf, S.Hallem, D.Engler. (2001). An Empirical Study of Operating System Errors. *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP)*.

- [118] Cplusplus. (2008). *Cplusplus*. Retrieved 2009, from <http://www.cplusplus.com/doc/tutorial/preprocessor.html>
- [119] L.P.Barreto, G.Muller. (2002). Bossa: A Language-Based Approach for the Design of Real Time Schedulers. *Proceedings of the 23rd IEEE Real-Time Systems*.
- [120] L.P.Barret, R.Douence,G.Muller,M.Sudholt. (2002). Programming OS Shedulers with Domain-Specific Languages and Aspects:New Approaches for OS kernel Engineering. *Workshop on Aspects, Componenets,and Patterns for Infrastructure Software at AOSD*.
- [121] C.Consel and R.Marlet. (1998). Architecting software using a methodology for language development. *Proceeding of the 10th International Symposium on Programming Languages,Implementations, Logics and Programs (PLILP/ALP)*.
- [122] A.Rashid, R.Ghitchyan. (2003). Persistence as an Aspect. (pp. 120 - 129). Boston, MA USA: ACM.
- [123] K.Mens, C.Lopes, B.Tekinerdogan, G.Kiczales. (1997). Aspect-Oriented Programming Workshop Report. *ECOOP Workshop Reader. LNCS 1357*. Springer-Verlag.
- [124] J.Suzuki, Y.Yamamoto. (1999). Extending UML for Modelling Reflective Software Components. *International Conference on the Unified Modelling Language (UML)*.
- [125] S.Clarke. (2000). Designing Reusable Patterns of Cross-Cutting Behaviour with Composition Patterns. *OOPSLA Workshop on Advanced Separation of Concerns* .
- [126] D.Holmes, J.Noble, J.Potter. (1998). Towards Reusable Synchronisation for Object-Oriented Languages. *ECOOP Workshop on Aspect-Oriented Programming*.
- [127] S. Clarke, R.J. Walker. (2001). Composition Patterns: An Approach to Designing Reusable Aspects. *ICSE*.
- [128] A.Rashid. (2000). On to Aspect Persistence. *GCSE Syrup. LNCS 2177*, pp. 26-36. Springer-Verlag.
- [129] A.Rashid. (2002). Weaving Aspects in a Persistent Environment. *ACM SIGPLAN Notices*. 37, pp. 36 - 44. ACM.
- [130] R.G.G.Cattell, D.Barry, M.Berler, J.Eastman, D.Jordan, C.Russel, O.Schadow, T.Stenienda, F.Velez. (2000). *The Object Data Standard: ODMG 3.0*. Morgan Kaufmann.

- [131] J.Kienzle, R. Guerraoui. (2002). AOP: Does It Make Sense? The Case of Concurrency and Failures. *ECOOP. LNCS 2374*, pp. 37-61. Springer-Verlag.
- [132] Sun Microsystems. (2008). *Java Tutorials: The reflection API*. Retrieved 2009, from <http://java.sun.com/docs/books/tutorial/reflect/index.html>
- [133] D. Parsons, A. Rashid, A. Speck, A. Telea. (1999). *A 'Framework' for Object Oriented Frameworks Design TOOLS Europe*. CS Press.
- [134] A. Rashid, P. Sawyer. (1999). Dynamic Relationships in Object Oriented Databases: A Uniform Approach. *DEXA, LNCS 1677*, 26-35.
- [135] A. Rashid. (2001). A Hybrid Approach to Separation of Concerns: The Story of SADES. *Reflection conference. LNCS 2192*, pp. 231-249. Springer-Verlag.
- [136] B. Silva, E. Figueiredo, A. Garcia, D. Nunes. (2008). Refactoring of Crosscutting Concerns with Metaphor-Based Heuristics. *In Electronic Notes in Theoretical Computer Science*.
- [137] I. Kiselev. (2002). Aspect-Oriented Programming with Aspect J: SAMS.
- [138] J. Araujo, A. Moreira, I. Brito, and A. Rashid. (2002). Aspect-Oriented Requirements with UML. *International Conference on Unified Modelling Language UML*.
- [139] A. Rashid, P. Sawyer, A. Moreira, and J. Araujo. (2002). Early Aspects: A Model for Aspect-Oriented Requirements Engineering. *Proceedings of IEEE Joint International Conference on Requirements Engineering (RE)* (pp. 199-202). IEEE Computer Society.
- [140] I. Jacobson, P. Ng. (2005). *Aspect-Oriented Software Development with Use-Cases*. Addison-Wesley.
- [141] IBM. (2010). *IBM WebSphere*. Retrieved 2010, from IBM WebSphere: <http://www-01.ibm.com/software/websphere/>
- [142] IBM. (2010). *TAM*. Retrieved 2010, from TAM: <http://www-01.ibm.com/software/tivoli/products/access-mgr-e-bus/>
- [143] IBM. (2010). *Tivoli Identity Manager*. Retrieved 2010, from Tivoli Identity Manager: <http://www-01.ibm.com/software/tivoli/products/identity-mgr/>
- [144] UML. (2010). *UML*. Retrieved 2010, from UML: <http://www.uml.org/>

- [145] AOSD-Europe. (2002). *European Network of Excellence on Aspect-Oriented Software Development*. Retrieved 2005, from <http://www.aosd-europe.net/>
- [146] C.Lopes, S.Bajracharya. (2005). An Analysis of Modularity in Aspect Oriented Design. *International Conference on Aspect-Oriented Software Development (AOSD)* (pp. 15-26). ACM.
- [147] M.A.Kersten,G.C.Murphy. (1999). Atlas:A Case Study in Building a Web-based Learning Environment using Aspect-oriented Programming. *OOPSLA. 34(10)*. ACM, SIGPLAN Notices.
- [148] M.Lippert, C.V.Lopes. (2000). A Study on Exception Detection and Handling Using Aspect-Oriented Programming. *22nd International Conference on Software Engineering*, (pp. 418-427).
- [149] J.Hannemann,G.Kiczales. (2002). Design Pattern Implementation in Java and AspectJ. *Proceedings of OOPSLA*. ACM.
- [150] N.Cacho, C.Sant'Anna, E.Figueiredo, A.Garcia, T.Batista, C.Lucena. (2002). Composing Design Patterns: A Scalability Study of Aspect-Oriented Programming. *International Conference on Aspect-Oriented Software Development (AOSD)* (pp. 109-12). ACM.
- [151] A.Garcia, C.Sant'Anna, E.Figueiredo, U.Kulesza, C.Lucena, A.v.Staa . (2005). Modularizing Design Patterns with Aspects: A Quantitative Study. *International Conference on Aspect-Oriented Software Development (AOSD)* (pp. 3-14). ACM.
- [152] M.Brunting, A.v.Deursen, M.D'Hondt, T.Tourwe . (2007). Simple Crosscutting Concerns are not so Simple: Analysing Variability in Large-Scale Idioms-based Implementations . *International Conference on Aspect-Oriented Software Development (AOSD)* (pp. 199-211). ACM.
- [153] D112 - AOSD Europe. (2008). *Description of Advances and Integrations in the AO Language Design Space*. AOSD Europe.
- [154] D87 – Language Lab. (2007). *Initial design of advances and integrations in the AO language design space*. (C. (. K.Gybels (VUB), Ed.) AOSD-Europe.
- [155] D.B.Tucker, S.Krishnamurthi. (2003). Pointcuts and advice in higher-order language. *AOSD 2003 Conference*, (pp. 158–167).
- [156] D. Parsons, A. Rashid, A.Speck, A.Telea. (1999). *A Framework for Object Oriented Frameworks Design TOOLS Europe*. CS Press.

Appendix A

The following Table provides an overview of the language and execution models described in the survey. [50, p. 13]

Aspect-oriented Language	Language Survey	Execution Survey
Alpha	√	
AO4BPEL	√	√
AspectC++	√	√
AspectCOBOL	√	
AspectJ	√	√
AspectS	√	√
AspectWerkz	√	√
CaesarJ	√	√
CAM/DAOP	√	√
CARMA	√	
Compose*	√	
DemeterJ	√	
EAOP	√	√
FuseJ	√	√
HyperJ	√	
JAC	√	√
JAsCo	√	√
JBOSS AOP	√	√
Lasagne	√	
Object Teams	√	
OReA	√	
PROSE	√	√
Reflex	√	
Sourceweave.net	√	√
Steamloom	√	√
SuperJ	√	
VEJAL	√	
Weave.net	√	√

Appendix B

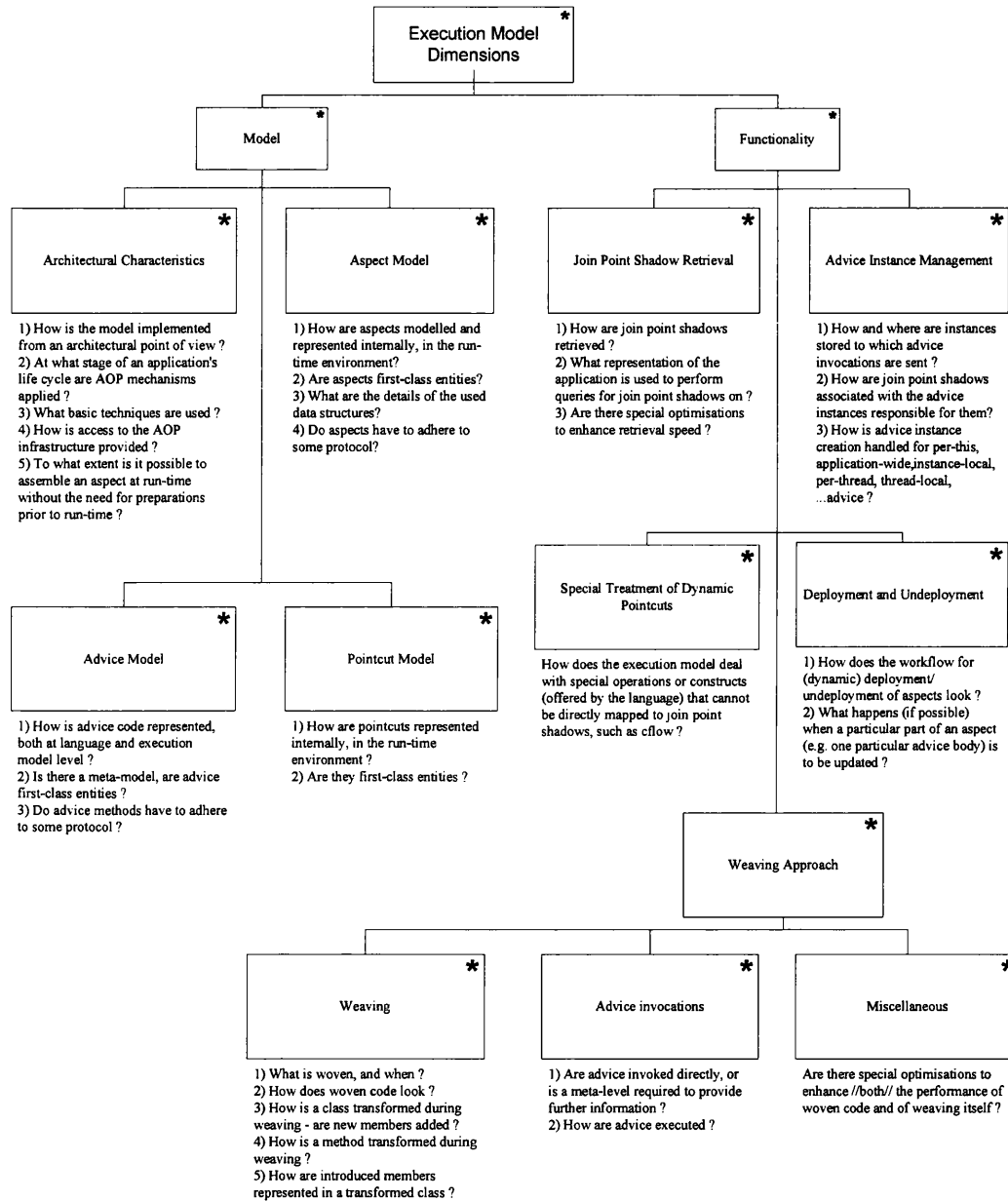


Figure 33 Execution Model Dimensions

As mentioned in section 3.3 that AOSD Languages Lab defined a set of questions regarding what the dimensions should be in agreement with all language lab partners. Figure 7 depicts the set of dimensions that were agreed and includes the related questions that define each dimension for the execution model [50, p. 14].